MASTER EN SCIENCES INFORMATIQUES

La proactivité dans l'Internet des Objets: réponses en temps réel et optimisations

Daloze, Florian

Award date:
2017

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2016–2017

La proactivité dans l'Internet des Objets : réponses en temps réel et optimisations.

Florian Daloze

Maître de stage : Denis Zampunieris

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Jean-Noël Colin

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Dans le contexte actuel d'émergence de l'Internet des Objets, il devient de plus en plus problématique de gérer, configurer et sécuriser tous les appareils qui nous entourent. Des solutions sont proposées par Intel et IBM pour gérer tout cet environnement de manière proactive ou autonome. Créer un moteur proactif demande de travailler les performances afin d'atteindre un logiciel capable de réagir en temps réel aux événements qui l'entourent. Ce mémoire présente différentes pistes de solutions permettant d'obtenir un système réactif et performant, tout en conservant une vision compatible avec le monde contraignant de l'Internet des Objets.

Abstract

In the current context of the emergence of the Internet of Things, it becomes more and more problematic to manage, configure and secure all the devices that surround us. Solutions are offered by Intel and IBM to manage this environment in a proactive or autonomous way. Creating a proactive engine requires working on performance in order to get a software capable of reacting in real time to the events that surround it. This thesis presents different ways of solutions to obtain a reactive and efficient system while maintaining a vision compatible with the binding world of the Internet of Things.

Avant-propos

Ce mémoire est la compilation d'heures de recherches et d'analyses, et il ne fut possible que grâce à une série d'acteurs que je tiens tout particulièrement à remercier.

Une grosse partie du travail fut accomplie lors de mon stage à l'Université de Luxembourg, au sein de l'équipe de M. Zampunieris, mon maître de stage, que je tiens tout spécialement à remercier pour son accompagnement, son implication dans la recherche et toutes les idées qu'il a pu partager au cours du stage.

Je voudrais aussi remercier Sandro Reis pour m'avoir fait découvrir les méandres du moteur proactif, Remus Dobrican et Gilles Neyens pour leur présence et l'apport de leur expérience au cours de mon travail.

Mes remerciements vont aussi à M. Jean-Noël Colin, mon promoteur, qui m'a donné d'excellentes idées et m'a soutenu durant la rédaction de ce mémoire.

Merci aussi à ma famille pour sa patience et son soutien, mes amis pour l'aide que nous nous sommes apportée mutuellement.

Sommaire

Table des figures

Chapitre 1

Introduction

L'informatique augmente son champ d'action d'année en année, de jour en jour, et se répand de plus en plus dans la vie de tous les jours et dans les objets du quotidien. Tous ces systèmes se multiplient et se complexifient. Il devient dès lors de plus en plus difficile de gérer tous ces appareils de manière efficace, que ce soit dans la maintenance, la compatibilité entre les appareils ou encore la sécurité. Un grand nombre de logiciels embarqués dans les objets connectés s'avèrent mal sécurisés, souvent suite à une mauvaise configuration.

Pour répondre à ces problèmes, Intel et IBM ont avancé le principe d'"autonomic computing"[10] et de "proactive computing"[21]. Ces pistes de solutions se dirigent vers une capacité d'auto-gestion, d'adaptation et d'anticipation de la part des systèmes. Il s'agit de portes ouvertes sur un monde où l'informatique serait de plus en plus indépendante et résiliente sur l'aspect technique, où les systèmes pourraient être tournés vers l'utilisateur.

A l'Université de Luxembourg, M Denis Zampunieris et son équipe ont développé un moteur en Java dont l'objectif est d'implémenter des scénarios de proactivité. Leur moteur pouvait ainsi réagir à des événements liés à la gestion administrative des étudiants de l'Université : inscription d'un étudiant, changement d'horaires, etc... Le moteur pouvait alors réagir sans avoir un accord humain et ainsi, par exemple, envoyer des documents informatifs aux étudiants ou leur notifier un changement de local.

Lors de mon stage à l'Université de Luxembourg, nous nous sommes concentrés sur les performances de ce moteur. En effet, qui dit proactivité, dit réactivité. Un système qui réagit bien, mais qui réagit trop tard, perd de son utilité. L'objectif était donc d'obtenir un moteur qui assure une prise en charge rapide d'un événement, où tous les éléments inutiles pouvant ralentir l'exécution d'un scénario sont supprimés.

Cette recherche est passée par plusieurs étapes, et chacune d'elle a fait l'objet d'analyses et de tests afin de voir quelles sont les combinaisons d'algorithmes et d'architectures qui peuvent permettre une telle réactivité, tout en gardant le principe de base du moteur précédemment développé. Les deux plus grandes pistes explorées furent l'ordonnancement des événements et le parallélisme des calculs. D'autres améliorations plus proches du code ont aussi été effectuées.

Ce mémoire présente donc les différentes pistes en question et évalue l'intérêt et les risques de chaque solution, pour enfin mettre tous ces éléments ensemble.

Dans un deuxième temps, nous avons voulu rendre le moteur capable d'entrer dans le monde de l'Internet des Objets. D'une part, une attention toute particulière a été portée

sur la consommation du logiciel, afin qu'il puisse être embarqué sur des appareils légers, malgré des contraintes de batterie ou de puissance. Cet aspect a été gardé tout au long du développement. D'autre part, nous avons recherché comment intégrer un ou plusieurs protocoles réseaux de l'Internet des Objets et le(s) faire fonctionner en collaboration avec le moteur.

Toutes ces recherches nous ont permis d'obtenir un moteur plus réactif, mais beaucoup d'autres améliorations sont encore possibles, au niveau de la proactivité comme des communications entre moteurs par exemple.

Chapitre 2

Etat de l'art

2.1 Proactive computing

2.1.1 Contexte

Depuis les débuts de l'informatique, il a toujours été question de rendre les machines et leur fonctionnement accessibles et compréhensibles. Pour ce faire, un travail méticuleux a été et est effectué tous les jours sur l'interface des différentes applications et sur la manière dont nous pouvons interagir avec elles, que ce soit en termes graphiques ou en termes d'outils (souris, claviers, gants, casques...). Toutefois, malgré le fait que cet aspect reste nécessaire, la multiplicité grandissante des appareils qui nous entourent nous amène à l'impossibilité de gérer tous ceux-ci si on conserve l'idée que chaque action d'une machine doit être demandée par un humain. C'est en effet à ce moment qu'apparaît la nécessité d'avoir un environnement technologique proactif, où chaque machine serait capable de gérer ses données et tâches avec le moins d'interactions possible avec l'utilisateur. De plus, non seulement le nombre de machines est croissant, mais elles sont infatigables, elles peuvent travailler 24h/24h sans interruption, et aucun être humain ne peut rester disponible tout ce temps pour interagir avec elles.

La première vision d'un ordinateur travaillant comme partenaire autonome de l'homme est apparue en 1960 dans "Man-computer symbiosis" de J.C.R. Licklider [12]. Dans cet article, M. Licklider explique que l'informatique jusque là se résumait à un outil pour l'homme, qui lui permet d'aller plus vite pour exécuter certaines tâches, comme une "extension de son bras". Mais, déjà à cette époque, il évoque la possibilité que les machines puissent devenir plus qu'un outil passif. Il entrevoit plusieurs problèmes à une possible proactivité : tout d'abord que l'informatique ne devra plus répondre à des questions, mais être capable de trouver les questions pertinentes, et que cela demandera aux machines une aptitude à réagir en temps réel, ce qui était impossible au vu des performances de l'époque.

En 1990, des systèmes basés sur la collaboration commencent à apparaître [18]. En effet, les systèmes informatiques grossissent, et la gestion de l'ensemble devient compliquée. On parle alors de logiciels d'aide à la décision, ou de monitoring de processus.

Au début des années 2000 apparaît la nécessité d'avoir des logiciels conscients du contexte et de leur environnement. Les systèmes ne doivent plus aider l'utilisateur uniquement dans ses calculs, mais aussi être conscients du contexte physique afin d'adapter leurs décisions dans un cadre plus large [9]. Là où un système était incapable de réagir si son contexte changeait, un système proactif se veut capable d'adaptation et d'anticipation.

Car en effet, une nouvelle vision de la gestion des systèmes informatiques s'impose, et, sur ce point, deux écoles proposent une vision différente de la situation : Intel et IBM.

IBM propose sa vision de l'autonomic computing, qui s'oriente vers la résistance et la robustesse des applications grâce à une capacité d'adaptation. Intel, de son côté, propose une solution de proactive computing, où la machine serait capable d'anticiper et de prendre des décisions en fonction du contexte.

Ainsi, David Tennenhouse (Intel) soulève trois caractéristiques [21] que les systèmes proactifs se devront de respecter :

1. Getting physical : les systèmes proactifs seront intimement connectés au monde qui les entoure en utilisant des senseurs et des actuateurs pour à la fois superviser et façonner leur environnement. Les recherches explorent le couplage entre les systèmes réseaux et leur environnement.
2. Getting Real : les ordinateurs proactifs vont répondre régulièrement à des stimuli externes à une vitesse plus rapide que ce que l'humain peut gérer. Les recherches dans ce domaine essaient de réduire le fossé entre la théorie du contrôle et l'informatique.
3. Getting out : les ordinateurs interactifs placent délibérément les humains dans le processus. Néanmoins, la diminution du temps disponible implique des recherches du côté d'une proactivité plaçant l'homme au-dessus du processus.

Cette anticipation, qui est aussi une prise de décision, doit toujours être en accord avec les utilisateurs. Un choix est rarement binaire et l'ajustement peut être relatif aux envies humaines. Un gestionnaire de la température dans une maison ne pourra jamais déterminer avec exactitude la température à avoir dans chaque pièce, même si on lui fournit agenda, température préférée, ou toute autre donnée pouvant aider sa décision [21].

Ces deux visions peuvent se compléter, mais toujours est-il que l'informatique doit évoluer, car l'être humain ne peut plus gérer la puissance de calcul ni le flux d'informations disponibles et, dans le futur, il sera nécessaire de laisser une part de ce traitement aux machines, pour se concentrer sur la supervision.

2.1.2 Autonomic computing

IBM publie en 2001 un manifeste montrant les premiers signes d'une crise de la complexité des logiciels [10]. Il le montre par des applications gigantesques de milliers de lignes de code nécessitant des techniciens compétents pour l'installation, la maintenance et la configuration. Il n'est plus possible de maintenir manuellement des systèmes complexes, eux-mêmes composés de sous-systèmes, et la multiplicité des connexions à Internet ne fait que renforcer le problème. De plus, plus la complexité augmente, plus l'anticipation devient difficile pour les architectes logiciels. On estime maintenant qu'environ un tiers à une moitié du budget d'une entreprise est dépensé à prévoir ou réparer des bugs [6]. Pour répondre à certains problèmes dus à cette croissance, IBM propose donc l'autonomic computing, qui suit huit principes :

1. self-monitoring.
2. self-heal.
3. self-configure.
4. amélioration des performances.

5. connaitre son environnement.
6. se défendre contre les attaques.
7. communiquer avec des standards ouverts.
8. anticiper les actions de l'utilisateur.

[22]

Ces huit principes sont applicables aussi bien à un système particulier qu'à un ensemble de systèmes communiquant entre eux. Lorsque Paul Horn, vice-président de la recherche chez IBM, a proposé le terme d'"autonomic computing", il a fait un lien avec la biologie, qui illustre très bien ce qui était recherché. Dans le corps humain, la circulation sanguine, l'appareil digestif, les défenses immunitaires sont autonomes : ils nous permettent de rester en vie en travaillant sans que nous devions nous en occuper à chaque instant. Ainsi, notre cerveau (du moins la partie consciente) n'est pas constamment préoccupé par toutes ces tâches "bas niveaux" [10].

La partie mise en avant dans les systèmes autonomes est l'auto-gestion. Le système doit continuellement s'auto-surveiller, et pour ce faire, IBM cite régulièrement quatre points importants, que vous pourrez retrouver dans le tableau 2.1[10]. Voici quelques explications supplémentaires sur ces quatre éléments.

Table 1. Four aspects of self-management as they are now and would be with autonomic computing.

Concept	Current computing	Autonomic computing
Self-configuration	Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmers weeks.	System automatically detects, diagnoses, and repairs localized software and hardware problems.
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures.

FIGURE 2.1 – Les 4 principaux aspects de l'autonomic computing [10]

2.1.2.1 Self-configuration

La configuration de gros logiciels ou d'ensemble de logiciels peut prendre un temps et des ressources considérables. Des systèmes tels que SAP peuvent nécessiter des dizaines d'experts travaillant pendant des mois pour arriver à un logiciel personnalisé et stable. L'idée derrière l'auto-configuration est la capacité du système à s'adapter à des besoins de haut niveau. Les composants sont capables de s'adapter entre eux et le logiciel va prendre en compte les machines sur lesquelles il est installé [10].

2.1.2.2 Self-optimization

Certains logiciels complexes, comme WebSphere ou des bases de données comme Oracle ou DB2, possèdent une pléthore de paramètres qui influent sur les performances du système global. Le choix adéquat de ces paramètres pour atteindre un niveau d'optimisation optimal peut changer régulièrement, or leur modification prend du temps et nécessite une bonne maitrise du logiciel. Un système capable de s'auto-optimiser pourrait mettre à jour les options nécessaires en fonction des besoins de manière proactive en analysant les changements survenus dans le système en continu [10].

2.1.2.3 Self-healing

Les grosses entreprises en IT ont souvent des équipes entières dédiées au tracking et à la correction de bugs. Ces opérations peuvent parfois prendre des semaines pour finalement voir le bug disparaître sans même avoir pu être identifié. Les systèmes autonomes vont quant à eux détecter, diagnostiquer et réparer des problèmes localisés provenant de bugs logiciels ou hardwares. En utilisant les données de configuration connues du programme, une analyse peut être faite notamment grâce aux logs ou à d'autres données collectées pour trouver la source des problèmes. Le système va alors pouvoir réagir et tenter de corriger lui-même le problème [10].

2.1.2.4 Self-protection

Malgré qu'il existe des pare-feux et des moyens de défenses multiples, c'est toujours l'être humain qui doit décider comment protéger son infrastructure. Un système autonome peut se protéger de deux manières : tout d'abord, se défendre contre les attaques qui peuvent entraîner des dégâts en cascade que la capacité d'auto-réparation ne peut corriger, et ensuite, il peut aussi essayer d'anticiper et d'adapter le système en fonction des différents rapports à sa disposition [10].

2.1.3 Proactive computing

M. David Tennenhouse parle pour la première fois des systèmes proactifs en 2000 dans son article "Proactive Computing", paru dans le magazine "Communication of the ACM"[21]. Il est alors question d'un système qui ne place plus l'être humain dans le processus comme une personne qui donne ses instructions, mais plutôt comme un superviseur. Il énumère ainsi sept principes capitaux aux systèmes proactifs [21] :

1. Connecté au monde physique : afin de pouvoir être proactif et d'anticiper des événements à venir, un système se doit d'être informé au mieux des changements survenant dans l'environnement de l'utilisateur. Dans un monde où nos données sont partagées dans un réseau mondial de serveurs, le traitement de ces données va requérir une information personnalisée et propre à la situation de la personne en question. M. Tennenhouse cite ainsi plusieurs exemples : gestion du trafic routier, étude de micro-climat, localisation de personnes lors d'un tremblement de terre...
2. Le système doit être interconnecté et capable de s'adapter à différents niveaux de réseau. Un système proactif pourrait avoir à gérer un ensemble de systèmes proactifs, eux-mêmes gérés par un autre système. La communication est importante, pour renforcer le premier point, et ainsi agir à des échelles différentes.
3. Le système doit pouvoir gérer le macro traitement.
4. Être capable de composer avec l'imprévu et l'aléatoire. Le monde qui nous entoure a toujours sa part d'imprévisibilité, comme nous le montre bien la théorie du chaos et son exemple typique, à savoir la météo : il est toujours impossible actuellement

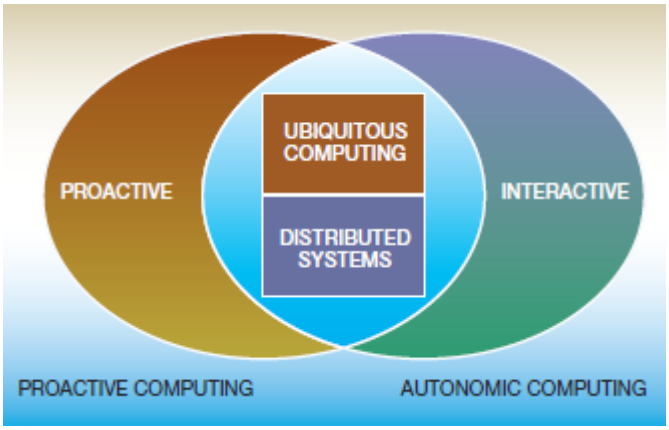

FIGURE 2.2 – La relation entre les deux paradigmes de programmation [22]

de prédire avec un taux de certitude de 100% s'il pleuvra ou non dans un mois. Dès lors, tout un pan de recherches s'ouvre sur les réactions possibles d'une machine face à l'incertitude et aux données manquantes.

5. Anticiper. Malgré que le monde ait sa part d'imprévisibilité, certaines choses peuvent être raisonnablement anticipées, et la proactivité nécessite cette part d'anticipation, pour prendre les devants et proposer au mieux les actions à effectuer.
6. Réponse en temps réel en boucle fermée. Il est nécessaire de penser à la rapidité des systèmes proactifs et à la vitesse à laquelle une réponse va pouvoir être apportée à un stimulus ou une absence de stimuli. Toutefois, malgré cette réponse en temps réel, il demeure important que l'homme reste dans la boucle en tant que contrôleur.
7. Rendre le système personnel. Comme pour l'autonomic computing, le système doit pouvoir s'adapter à l'installation actuelle, à son environnement et au besoin de l'utilisateur spécifique.

2.1.4 Proactive and autonomic systems

Les systèmes autonomes et proactifs sont différents mais permettent tous les deux d'améliorer la gestion des systèmes complexes et d'élargir les possibilités de l'informatique future. D'une part, l'autonomie va assurer la résilience des systèmes et la facilité d'utilisation, tandis que la proactivité va donner un espace supplémentaire à l'anticipation et à une aide en faveur de l'utilisateur. Nous pouvons voir dans la figure 2.2 la relation entre les deux paradigmes.

2.1.5 La proactivité et l'Internet des objets

Comme nous l'avons vu plus haut, implémenter la proactivité dans les systèmes qui nous environnent nécessite de demander à ces systèmes d'être capables de capter les changements dans leur environnement (pas que le leur, mais surtout l'environnement de l'être humain qui utilise ce système). Cela s'inscrit très bien dans l'émergence de l'Internet des objets [2].

Les environnements connectés vont pouvoir faciliter l'implémentation d'un système proactif, en permettant à ce dernier de préciser et d'ajuster ses décisions avec une plus grande quantité d'informations. On imagine ainsi facilement dans le cadre des smart-homes une gestion proactive de l'électricité, basée sur l'analyse des comportements des utilisateurs. Ou encore la possibilité de préparer de l'eau chaude à l'avance pour le déjeuner de l'utilisateur par exemple. Dans des contextes plus professionnels, on peut très bien

imaginer une gestion du personnel proactive à travers les différents outils mis à disposition : pointage, machine à café... Au niveau d'une ville aussi, tout est possible : gestion de la circulation, des flux de piétons ou de voitures, des places de parking... Implémenter une proactivité dans la gestion de ces scénarios va demander une foule d'informations sur l'environnement du système, et c'est là que l'Internet des objets va montrer son intérêt. Il existe maintenant une foule de capteurs différents, et le challenge actuel est de les faire tous fonctionner ensemble. La combinaison de toutes ces informations peut permettre alors de créer une plus-value pour l'utilisateur, par exemple grâce à une gestion proactive.

Mais non seulement l'Internet des objets va pouvoir aider la proactivité, mais l'inverse est vrai aussi. Actuellement l'Internet des objets rencontre de gros problèmes structurels, en grosse partie liés à la sécurité des objets connectés. On peut déjà trouver des articles parlant d'une fin possible d'Internet si aucune mesure n'est prise [13]. La proactivité peut alors aussi aider à l'installation et à la mise à jour des logiciels embarqués.

2.1.6 Systèmes proactifs existants

Dans les paragraphes qui suivent se trouvent des exemples de logiciels ou entreprises existantes qui ont une gestion proactive de leur système.

2.1.6.1 PureStorage

PureStorage [15] est une société qui propose des solutions de stockage avec une gestion proactive de son infrastructure. Cette gestion permet à la société d'anticiper les problèmes pouvant survenir sur une de ses machines. Ainsi, lors d'un contact entre le client et PureStorage au sujet d'un problème technique, c'est le technicien qui est à l'origine de l'appel dans 58% des cas.

Son système permet ainsi :

1. d'anticiper les vulnérabilités susceptibles de poser problème ;
2. de détecter les changements d'environnement et de configuration ;
3. de détecter des problèmes de performance et de capacité

2.1.6.2 HP Proactive Care

HP propose une suite logicielle permettant de contrôler une infrastructure afin de l'améliorer en temps réel de manière proactive. Son système utilise les alertes de pré-défaillance, une analyse en temps réel basée sur la configuration actuelle et permet d'améliorer les performances et de bénéficier d'une assistance anticipative [7].

2.1.6.3 Proactive Parallel Suite

Proactive Parallel Suite est un logiciel développé par M. Denis Caromel, professeur à l'Université Nice-Sophia-Antipolis et repris plus tard par des membres de l'équipe OASIS à l'Institut National (français) de Recherche en Informatique et en Automatique. Ce logiciel est un outil de gestion de charge de travail. Grâce à un jeu d'exécutables ou de scripts, il peut répartir la charge de travail et exécuter en parallèle les différentes tâches. Le système possède une interface RESTful pour communiquer avec les entreprises. La figure 2.3 montre les différentes exécutions possibles d'une série de ressources. Le logiciel peut exécuter ses tâches en séquentiel, en parallèle ou de manière distribuée. [3] [23]

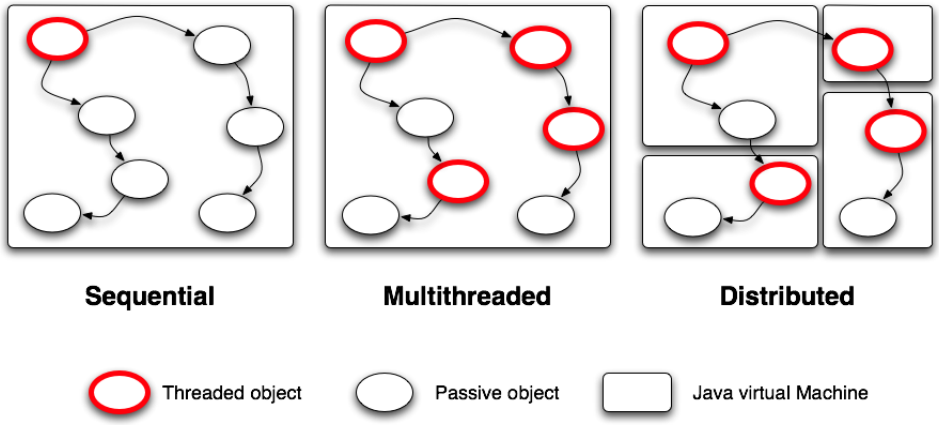

FIGURE 2.3 – Exécution de différentes ressources actives [23]

2.1.7 Moteur proactif existant

Un moteur proactif générique a été développé à l'Université du Luxembourg par l'équipe de M. Denis Zampunieris. Voici une présentation de ce moteur, qui sera étudié plus tard.

2.1.7.1 Présentation du moteur développé à l'Université de Luxembourg

Dans le cadre de recherches sur l'amélioration de l'e-learning à l'Université de Luxembourg, M. Denis Zampunieris et son équipe ont été amenés à développer un moteur proactif, destiné à répondre aux besoins du système. "Le moteur est théoriquement capable d'aider et d'assister les e-étudiants en respectant une liste de procédures précédemment définies - appelées scénarios proactifs. Cela étant, le système est capable de détecter un comportement "anormal" d'un étudiant et d'en communiquer les détails au professeur concerné ; ou alors le système peut vérifier les besoins d'un étudiant et y répondre par lui-même."[16]

2.1.7.2 Définition de scénarios

Pour implémenter leur système proactif, l'équipe de M. Zampunieris est partie du constat que la proactivité permet de répondre à des objectifs que l'on peut préciser. Dès lors, il est possible de créer des scénarios pour lesquels on va pouvoir définir des comportements à adopter. En effet, quand bien même le système doit pouvoir s'adapter à un environnement changeant et anticiper des scénarios, il est tout à fait possible de catégoriser ces changements et dès lors de préparer des marches à suivre pour ces événements. Rappelons que la proactivité n'implique pas ici d'intelligence artificielle et que donc il va être nécessaire d'anticiper le plus possible d'événements. En effet, le moteur a été conçu pour permettre une programmation différente, basée sur des règles et des scénarios. Le développement de ces éléments peut permettre de coder des scénarios réactifs ou proactifs, mais pas nécessairement capables d'apprendre par eux-même. Deux types de scénarios ont été identifiés : les Meta-scenarios et les Target-Scenarios.

Meta-Scenarios Les Meta-Scenarios correspondent à ce qu'on pourrait appeler le centre de détection. Il s'agit des scénarios qui vont détecter un événement (ou absence d'événement) et prendre une décision en conséquence. Ce sont ces scénarios qui analysent l'environnement du système en continu, afin d'être réactifs en cas de changement et de décision à prendre. Lorsqu'un événement a été détecté par un Meta-scenario, celui-ci va

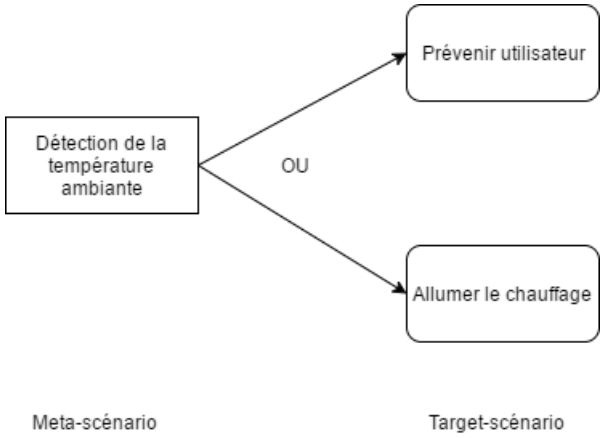

FIGURE 2.4 – Un Meta-Scénario pouvant choisir entre deux actions

alors prendre une décision sur le type de traitement à effectuer et lancer l'exécution d'un Target-Scenario, qui se contente d'exécuter la tâche. Tout le pan de la détection et de la décision revient donc aux Meta-scenarios, tandis que l'exécution de l'action revient aux Target-scenarios.

Target-Scenarios Les Target-scenarios sont exécutés par les Meta-scenarios. Ces scénarios ont pour objectif de répondre aux différents besoins relatifs aux événements détectés par les Meta-scenarios. Typiquement, ces Target-scenarios peuvent être autant de réponses différentes à un Meta-scenario. C'est ce dernier qui choisit quel action exécuter parmi les différentes possibilités qui s'offrent à lui.

2.1.7.3 Implémentation des scénarios

Après avoir défini des scénarios, il est nécessaire de pouvoir les exécuter. Pour ce faire, il faut détecter l'événement correspondant, l'identifier, choisir une marche à suivre et enfin réagir à cet événement. Pour ce faire, il a été choisi d'implémenter un moteur basé sur des règles (Rule Running System - RRS). Pour arriver à cette construction, l'idée était de partir sur le fonctionnement d'un processeur, qui exécute une série d'instructions, de préférence les plus courtes possibles afin d'en exécuter un maximum.

Les composants du moteur proactif La figure 2.5 nous montre les différents éléments principaux du moteur proactif. D'un point de vue extérieur, il se trouve des connexions depuis le moteur vers d'autres moteurs proactifs, vers un interface utilisateur ainsi que vers d'autres applications. En interne, un gestionnaire de contexte récolte les différentes informations depuis des capteurs ou des bases de données et interagit avec un moteur de règles, destiné à exécuter les différents scénarios. Celui-ci est en relation avec différentes queues et un gestionnaire de notifications.

Des règles Une règle est une classe Java qui représente une action exécutée par le moteur. Une règle s'inscrit dans un scénario, que ce soit un Meta-Scénario ou un Target-Scénario. Concrètement, une règle implémente 5 fonctions :

```
protected abstract void dataAcquisition() ;
```

Cette première fonction est destinée à récupérer les informations nécessaires au fonctionnement de la règle à l'extérieur de celle-ci. C'est ici donc que la règle va pouvoir se

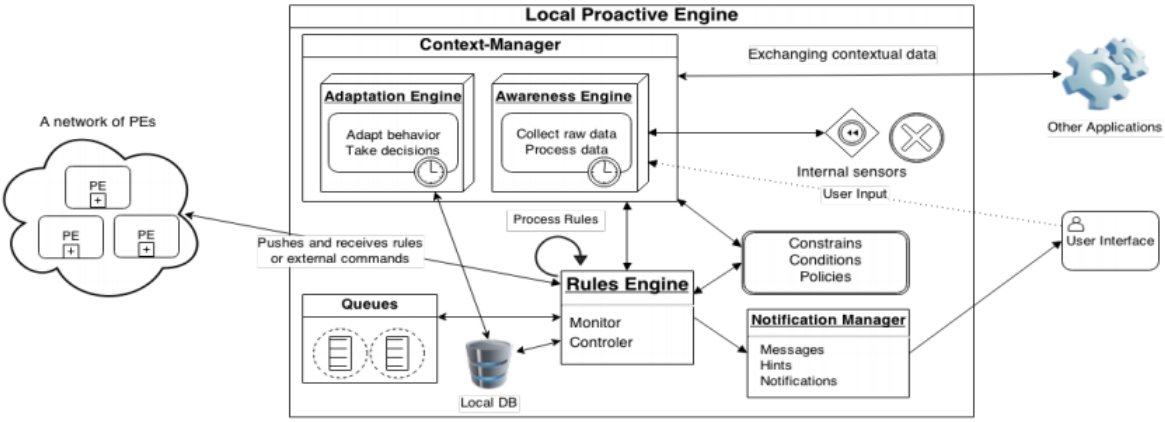

FIGURE 2.5 – L'architecture d'un moteur proactif [17]

connecter à une base de données, à un capteur, ou à tout autre élément sur lequel la règle doit inférer.

```
protected abstract boolean activationGuards() ;
```

Cette fonction permet de voir si les conditions nécessaires à l'exécution de la règle sont présentes. Il s'agit d'une suite de tests sur les variables préalablement récupérées dans la méthode dataAcquisition qui permet de savoir si le contexte a changé ou non et si la règle est pertinente dans le cas présent.

```
protected abstract boolean conditions() ;
```

Comme la précédente fonction, cette méthode permet d'exécuter une nouvelle série de tests afin de vérifier si on exécute la partie action de la règle ou non. La principale différence entre les deux méthodes est sémantique : si l'activationGuards n'est pas validée, l'exécution de la règle s'arrête net. Mais si le résultat est positif mais la condition ne l'est pas, "action" ne sera pas non plus activée, mais rulesGeneration sera bien exécutée.

```
protected abstract void actions() ;
```

Cette méthode est le noyau de la règle. C'est ici que la tâche pour laquelle la règle existe est effectuée. Pour que l'action puisse correctement se dérouler, les données nécessaires à son bon déroulement doivent avoir été récupérées dans la fonction "dataAcquisition".

```
protected abstract boolean rulesGeneration() ;
```

Cette dernière méthode n'est appelée que si "activationGuards" a bien été validée, et ce même si "conditions" est invalidée. A ce moment de l'exécution, la règle a fait (si nécessaire) sa partie "action" et son boulot est terminé. Toutefois, d'autres scénarios peuvent découler de cette règle. C'est donc ici qu'il est possible de générer de nouvelles règles.

Une liste d'entrée et une liste de sortie Le moteur possède deux listes : une liste d'entrée et une liste de sortie. Dans la liste d'entrée, on va trouver toutes les règles qui doivent se faire exécuter. Dans la liste de sortie, toutes les règles qui ont été générées par la fonction "rulesGeneration" lors de l'exécution des règles d'entrée.

Un noyau et des connexions Le reste du moteur est composé des différentes classes permettant d'exécuter les règles, d'établir des statistiques sur leur exécution, ou encore différentes fonctionnalités qui permettent d'établir des connexions vers des systèmes extérieurs, tels qu'Hibernate.

Les composants et leur fonctionnement

Une itération et un scénario Le moteur fonctionne sur un principe itératif. A intervalles réguliers, le moteur va prendre toutes les règles stockées dans la liste en entrée, exécuter les différentes règles et placer les règles générées dans la liste de sortie. Le moteur prend les règles dans l'ordre dans lequel elles arrivent. Dans une itération, on peut tout aussi bien exécuter des règles de Meta-Scénario comme des règles de Target-Scénario, sans distinction.

Dans un traitement typique d'un événement, l'exécution des règles relatives à ce scénario vont prendre au minimum deux itérations. En effet, une première itération va permettre d'activer la/les règles relatives au Meta-Scénario, et puis aux itérations suivantes la/les règle(s) relative(s) au Target-Scénario vont être exécutées. Dans ce calcul, nous omettons toutes les itérations ou une règle de Meta-Scénario ne se voit pas exécutée suite à une condition non remplie, mais qui se fait bien "vérifiée" à chaque itération.

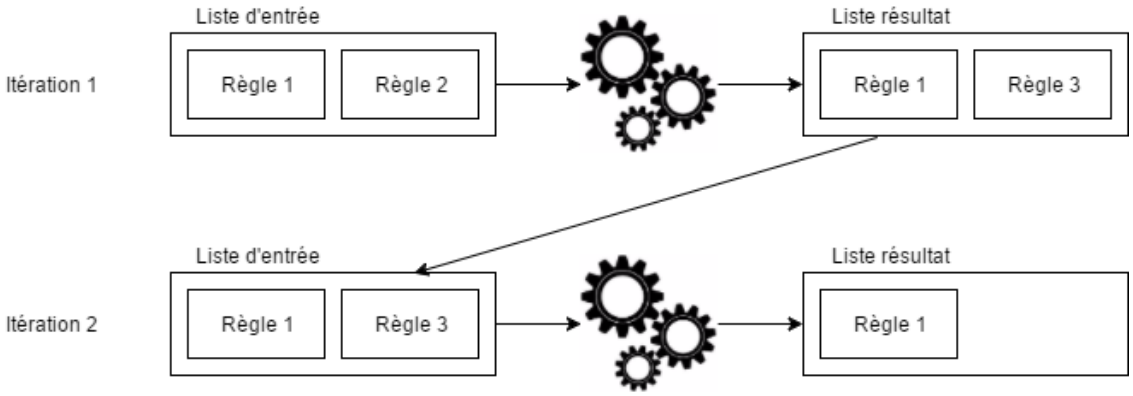

FIGURE 2.6 – Deux itérations d'exécution de règles

Dans la figure 2.6, nous illustrons le fonctionnement de deux itérations successives du moteur, sur un jeu de 3 règles. Les règles 1 et 2 sont des règles de Meta-Scénarios, tandis que la règle 3 est une règle de Target-Scénario. Lors de la première et de la deuxième itérations, la règle 1 ne produit pas de nouvelle règle et ne fait que se générer elle-même en liste de sortie. Cela signifie que la méthode "dataAcquisition" s'est déroulée normalement (car sinon, pas de génération de règle possible), mais que la condition est invalidée. Dans ce genre de cas, il est classique de copier la règle Meta-Scénario en sortie, afin de garder le scénario à l'écoute d'un événement qui le concerne.

Pour la règle 2, qui est donc elle aussi une règle Meta-Scénario, elle a vu sa condition validée à l'itération 1, ce qui a mené à la génération d'une règle Target-Scénario, la règle 3. Cette règle 3 va s'exécuter normalement à l'itération 2 et ne plus générer de règle, l'objectif étant atteint. Si le scénario de la règle 2 est un scénario qui doit être vérifié en permanence, il est tout à fait possible que la règle 2 ou 3 régénère une règle 2 en sortie d'exécution.

Gestion des ressources Par ressource, nous entendons ici tout élément servant de stockage de données utilisé par une règle du moteur. Ce peut donc être tout aussi bien une classe interne au moteur, qu'un élément extérieur. Le moteur étant synchrone, la gestion des données internes au moteur ne pose pas de problème, les accès à celles-ci se faisant naturellement, règle par règle. Abordons donc l'aspect plus intéressant des connexions extérieures.

De manière générale, les connexions vers des systèmes extérieurs sont ouvertes au lancement du moteur, et fermées à la fin de son exécution. Chaque règle a le droit d'accéder à ces ressources, mais les spécifications demandent à ce que ces requêtes soient placées à des endroits précis du code de la règle. Pour les accès de lecture, c'est la méthode "dataAcquisition" qui est dédiée. Les modifications de ressources doivent se faire suite à l'objet de l'action de la règle, et donc dans la méthode "action" de celle-ci.

Ceci peut malheureusement entraîner une certaine surcharge de la ressource extérieure. En effet, si les itérations se suivent sans interruption et qu'une itération ne voit pas de règle lourde s'exécuter, les itérations successives vont se contenter d'appeler toutes les "dataAcquisition" de chaque règle en continu. Pour peu que le moteur n'ait qu'une ressource extérieure, toutes les règles vont faire des appels en permanence sur cette ressource.

Afin de limiter l'impact de ces requêtes sur les ressources, un paramètre α a été introduit, qui régit le temps d'attente que le moteur doit respecter entre chaque itération. Ce paramètre devait être choisi afin d'avoir un équilibre entre la charge que la ressource peut supporter et la fréquence d'actualisation dont le moteur a besoin. Un autre paramètre N a lui aussi été créé afin d'indiquer un maximum de règles exécutables par itération. Ces deux paramètres sont à ajuster en fonction du système pour avoir un moteur équilibré. Dans la figure 2.7, chaque itération représente la liste des règles effectivement exécutées lors de l'itération en question. Ainsi, avec un paramètre N égal à 3, chaque itération voit un maximum de 3 règles exécutées, tandis que le paramètre α égal à 50 ms implique un délai équivalent entre l'exécution de chaque itération.

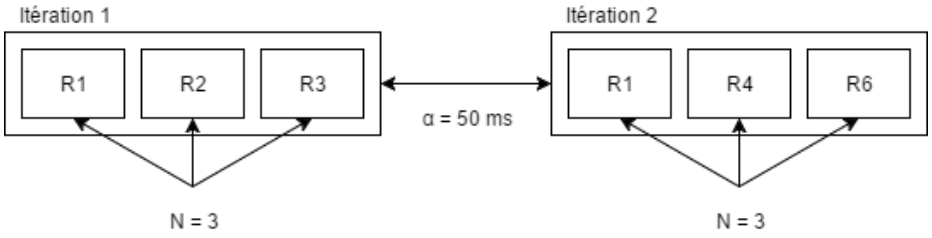

FIGURE 2.7 – Deux itérations et les paramètres associés

2.1.8 Vision de l'Iot

Le moteur tel qu'il a été développé peut facilement accepter des connexions depuis des ressources provenant de l'Internet des objets, moyennant une intégration des protocoles nécessaires par le programmeur. Par contre, le système n'est pas très adapté à une exécution sur un matériel plus faible. En effet, le moteur consomme une grande quantité de la ressource CPU disponible, qui peut être limitée dans de petites infrastructures, tout en ayant l'inconvénient de consommer énormément de batterie, qui elle aussi est généralement limitée.

2.2 La proactivité dans la gestion des ressources

De nos jours, il existe déjà plusieurs systèmes proactifs, principalement tournés vers la gestion de ressources. Internet grandissant, les serveurs sont de plus en plus nombreux et volumineux, et cette gestion proactive est déjà présente sur certains systèmes. Mais ici, la ressource est dans un sens plus large qu'une simple donnée (et c'est cette signification que nous garderons durant tout le mémoire), car une ressource peut être tout objet contrôlable par le moteur. Dans le document écrit par Antti Salovaara et Antti Oulasvirta [19], diverses méthodes sont proposées pour effectuer cette gestion proactive de ressources. En particulier, six actions distinctes sur les différentes périodes du cycle de vie d'une ressource sont passées en revue. La plupart de ces actions peuvent être vues comme des actions sensibles au contexte. Toutefois, on peut réellement leur donner une grande efficacité en intégrant de l'anticipation.

2.2.1 Préparer la ressource

Le premier point est la préparation d'une ressource de manière anticipative. Cette préparation doit toujours être signalée à l'utilisateur, et part de la prédiction que la ressource en question va être utilisée. Exemple : Une présentation doit avoir lieu dans le local X, or dans ce local le projecteur met 2 minutes à préchauffer. Le système peut donc anticiper le préchauffage et faire en sorte que le projecteur soit prêt au début de la présentation.

2.2.2 Optimiser la ressource

L'objectif dans l'optimisation de la ressource est de faire en sorte que l'utilisation de la part de l'utilisateur ait un effet maximum. De nouveau, le système doit être au courant des objectifs de l'utilisateur. Ce comportement est facilement applicable à la configuration d'un réseau, où le système peut essayer d'optimiser les routes à suivre et éviter les routes congestionnées.

2.2.3 Conseiller une utilisation de la ressource

Le système peut aussi aider l'utilisateur sur le choix de ressources à utiliser, et ainsi potentiellement proposer des ressources auxquelles il n'aurait pas pensé.

2.2.4 Manipuler la ressource

Lors d'une manipulation de ressources, le système peut aider l'utilisateur a effectuer sa tâche en lui donnant une dimension supplémentaire. Là où quelqu'un veut effectuer une action A, le système peut configurer l'environnement en fonction de ces changements et ainsi simplifier la tâche de l'utilisateur.

2.2.5 Contrôler l'utilisation de la ressource

Le système peut aussi contrôler la manière dont ses ressources sont utilisées et interdire ou autoriser son accès. Un exemple simple serait l'interdiction de démarrer le moteur d'une voiture si celle-ci détecte que son conducteur est saoul.

2.2.6 Finaliser l'utilisation d'une ressource

Après l'utilisation d'une ressource, un part d'anticipation peut être effectuée pour savoir que faire de la ressource. L'archiver, la laisser en veille ou la redémarrer ?

Chapitre 3

Problématique

En vue d'implémenter un moteur proactif répondant aux critères proposés par M. Tennenhouse, il est possible de repartir de ces différents critères et de trouver des premières pistes pour répondre à ces exigences :

1. Connecté au monde physique : idéalement, le moteur devrait être capable de se connecter à une multitude d'outils externes. L'idéal pour cela est probablement d'utiliser des standards de communication.
2. Le système doit être interconnecté et capable de s'adapter à différents niveaux de réseau : pour cela, le moteur proactif devrait pouvoir échanger des informations ou tâches avec d'autres moteurs proactifs travaillant à différents niveaux.
3. Le système doit être capable d'effectuer du macro traitement.
4. Être capable de composer avec l'imprévu et l'aléatoire : il va être important d'avoir, d'une part, une programmation robuste et, d'autre part, des scénarios englobant un large panel de situations.
5. Anticiper : de même que pour le point précédent, c'est la création des scénarios qui va être déterminante sur la capacité du système à anticiper un événement.
6. Réponse en temps réel en boucle fermée : faire en sorte que tout événement puisse être pris en compte en direct, et qu'une interaction avec le moteur soit possible pour un utilisateur.
7. Rendre le système personnel : il faut penser à un système configurable et adaptable.

La proactivité dans les logiciels est encore en phase de recherches et les pistes de solution pour répondre à ces exigences sont multiples et il est impossible de les traiter toutes ici. En particulier, la suite de ce document va se focaliser sur les performances du système et les liaisons que celui-ci peut avoir avec son environnement, en particulier dans le monde émergent de l'IoT. Afin de partir d'une base existante, la solution proposée ici utilisera le concept de scénarios (Meta et Target), ainsi que les règles qui permettent l'implémentation de ces scénarios du projet de M. Zampunieris et de son équipe.

3.1 Réponse en temps réel

Dans le moteur existant, il existe plusieurs éléments qui nuisent aux performances du système.

3.1.1 Des itérations et un paramètre α

Chaque exécution d'une itération du moteur est séparée par un laps de temps alpha configurable.Ce paramètre est destiné à alléger la charge que peut exercer le moteur sur ses ressources extérieures.

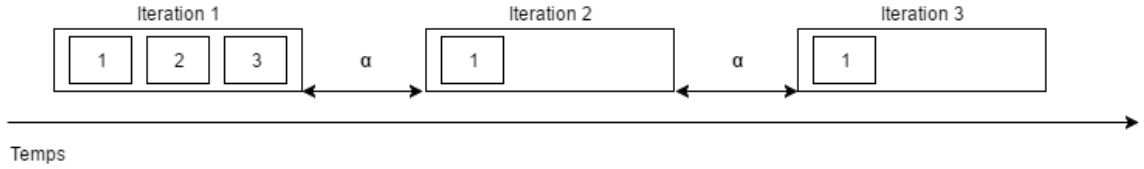

FIGURE 3.1 – Arrivée d'événements

L'utilisation de ce paramètre est problématique dans une situation où la réactivité est nécessaire. En effet, une règle survenant durant ce laps de temps ne sera pas traitée avant l'itération suivante.

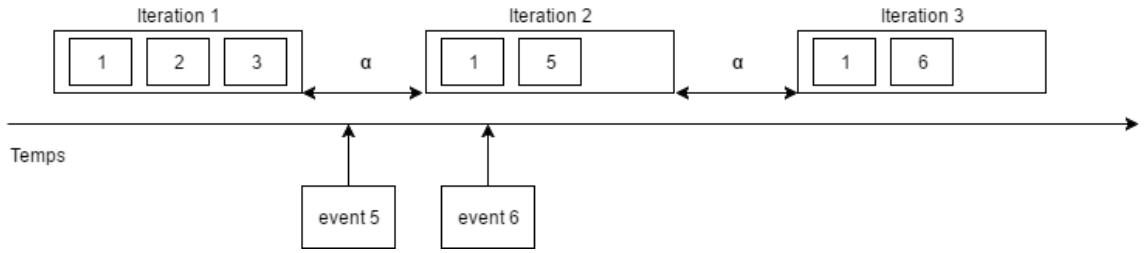

FIGURE 3.2 – Temps d'attente

Comme l'illustre ici la figure 3.2, l'événement numéro 5 doit attendre la fin du paramètre α pour être exécuté dans l'itération suivante et l'événement numéro 6 doit attendre la fin de l'itération actuelle plus le paramètre α avant d'être compris dans une itération.

3.1.2 Une exécution synchrone

Lors d'une itération, les règles sont exécutées les unes après les autres. Il est donc évident, que si une règle prend un temps considérable à s'exécuter, toutes les autres règles et tous les événements ne pourront être traités.

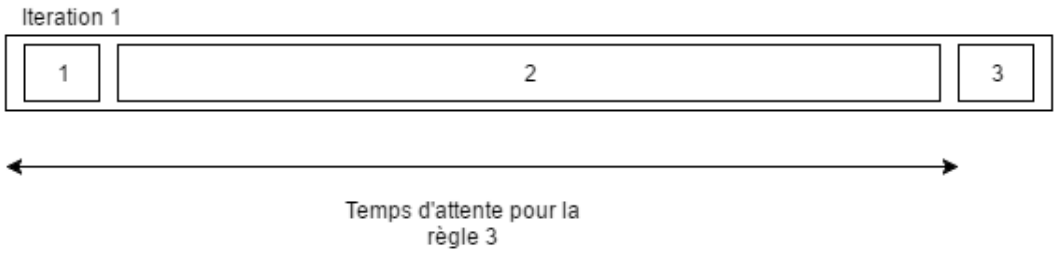

FIGURE 3.3 – Le paramètre alpha

Dans la figure 3.3, le scénario comprenant la règle 3 ne peut être traité tant que les règles 1 et 2 n'ont pas été exécutées. Or la règle 2 prend un temps considérable, qui se répercute sur la réactivité du système.

3.1.3 Une subdivision des scénarios en petites règles

Dans la vision de M. Zampunieris, une règle doit être la plus petite possible et ne représenter qu'une action logique. Un scénario sera donc représenté par une suite de règles

qui sont elles-mêmes une action ou décision à prendre. Toutefois, lorsqu'on désire de la réactivité, il nous faut gagner du temps sur deux points : le temps nécessaire pour que l'événement soit pris en compte et le temps nécessaire pour traiter l'événement. Or, si notre scénario est correctement subdivisé en multiples règles, il nous faudra autant d'itérations pour traiter le scénario, et autant de fois un temps α à attendre.

3.2 Questions

En partant de ces observations, une série de questions se posent.

Quels sont les moyens permettant d'améliorer la réactivité du moteur proactif, et quels sont les avantages et inconvénients de chacun ? Cette question découle directement des caractéristiques d'un moteur proactif données par M. Tennenhouse.

Tout en gagnant en réactivité, n'est-il pas possible d'alléger l'impact du moteur sur les ressources extérieures ? Par cette question, nous amenons l'idée d'essayer d'inverser le processus de notification. En effet, le moteur essaie par lui-même de se mettre à jour vis-a-vis de ses ressources extérieures, mais ne peut-on pas inverser le sens de notification ?

Outre le fait d'alléger le poids exercé sur les ressources extérieures, peut-on limiter la consommation du moteur lui-même tout en conservant de la réactivité ?

Chapitre 4

Exploration des solutions possibles

Avant de faire le choix de créer un moteur proactif asynchrone pour répondre au besoin de réactivité, il est intéressant de se pencher sur différentes alternatives.

4.1 Améliorer l'ordonnancement

Au lieu de partir sur un traitement asynchrone des scénarios, cette section examine une version synchrone du moteur dans laquelle l'ordonnancement des règles à exécuter aurait été travaillé au mieux. Ce choix d'améliorer l'ordonnancement vient de la présence de règles qui prennent un temps considérable à s'exécuter et bloquent l'exécution d'autres règles. Si la règle en attente était prioritaire, peut-être aurions-nous pu l'exécuter avant les autres règles afin de limiter l'impact de l'exécution séquentielle.

Voici un exemple avec 4 scénarios : deux scénarios Meta et leur scénario Target :

— Meta-scenario 1 : "Création d'un nouvel utilisateur". Ce scénario détecte la création d'un nouvel utilisateur et génère le scénario target "Création d'un contexte minimum".
— Meta-scenario 2 : "Un utilisateur veut se connecter". Ce scénario se déclenche lorsqu'un utilisateur désire se connecter au système. Il génère le scénario "Connexion acceptée".
— Target-scenario 3 : "Création d'un contexte minimum". Ce scénario crée un contexte minimum pour l'utilisateur dans le système. Cela requiert pas mal de travail : création de fichiers utilisateur, de base de données, etc... Ce scénario prend du temps à s'exécuter.
— Target-scenario 4 : "Connexion acceptée". Ce scénario répond favorablement au client souhaitant se connecter. Ce Target-Scénario est exécuté si le Meta-Scénario 2 a choisi d'accepter le client.

Afin de ne pas complexifier inutilement l'exemple présent, il faut considérer actuellement qu'un scénario est implémenté en une seule règle. Les numéros qui suivent se rapportent donc autant à une règle qu'au scénario auquel la règle appartient.

La figure 4.1 nous montre comment se présente une exécution typique de cet exemple.

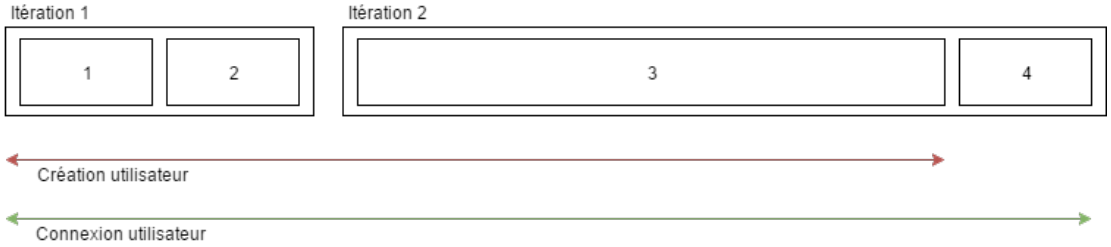

FIGURE 4.1 – Deux scénarios exécutés sur deux itérations

Dans cette illustration, la connexion est retardée par la création d'un nouvel utilisateur. Avec de l'ordonnancement, c'est le scénario de connexion qui est prioritaire à la création d'un nouvel utilisateur. En agissant ainsi, l'organisation de l'exécution des règles est semblable à l'illustration 4.2 :

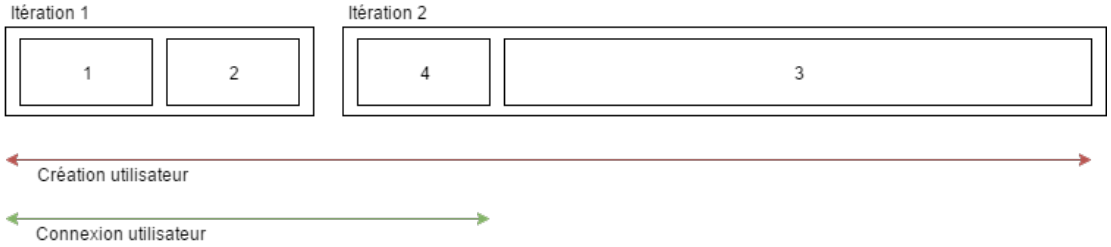

FIGURE 4.2 – Exécution de deux scénarios dans un contexte ordonnancé

Dans ce cas de figure, le temps considérable perdu par l'exécution de la règle 3 est évité. Ainsi, pour entièrement exécuter le scénario (règle 2 + 4), il faut attendre le temps d'exécution de la règle 1 + le paramètre α, ce qui devient acceptable, même s'il reste encore du temps à attendre.

Malheureusement, même si l'ordonnancement semble résoudre en partie le problème de performance, ce n'est en réalité pas le cas sur des scénarios plus complexes. En effet, l'ordonnancement permet d'éviter le blocage de la règle 3, mais ordonnancer l'itération 1 n'apporte rien, car le scénario doit dans tous les cas attendre l'itération 2 pour se terminer. Or, toutes les règles de l'itération 1 doivent être traitées pour passer à l'itération 2. En conclusion, l'ordonnancement dans la situation présente permet de gagner peu en réactivité, car il n'impacte que la dernière itération d'un scénario.

4.2 Un environnement événementiel

Au stade actuel, un problème plus intrinsèque au moteur se présente : faut-il garder le mécanisme d'itération ? En effet, comme il a été vu au point précédent, ce mécanisme bloque la mise en place d'un ordonnancement correct.

Il faut alors se poser la question suivante : pourquoi ces itérations existent, et qu'y trouve-t-on ?

De manière générale, il existe quatre types de règles dans une itération : des règles qui implémentent soit un scénario Meta, soit un scénario Target et qui sont soit "en attente", soit "en exécution".

Une règle d'un scénario Meta "en attente" : il s'agit d'une règle qui, comme son nom l'indique, implémente un scénario Meta. Par leur nature, les méta-scénarios sont là pour

réagir face à un événement prédéfini. Pour pouvoir capter ce dernier, la règle implémentant ce scénario va vérifier si cet événement s'est produit dans sa partie "condition". Pour que ce scénario réagisse relativement rapidement après que l'événement se soit produit, la règle va devoir vérifier sa condition le plus souvent possible, c'est-à-dire à chaque itération.

Évidemment, il est possible de raffiner cette analyse en constatant que certains scénarios peuvent se permettre d'être moins réactifs, mais cela sera abordé plus tard. Plus le moteur possède de scénarios, plus le nombre de règles avec une garde non validée (qui seront nommées "mortes" à partir de maintenant) qui sont exécutées à chaque itération est important. Il convient de bien rappeler que, même si la partie "action" d'une règle n'est pas exécutée, la méthode "dataAcquisition" l'est dans tous les cas. Cela implique que, même en cas de condition non validée, la règle a utilisé ses ressources et a donc bien un impact sur le moteur et son environnement.

Une règle d'un scénario Meta "en exécution" : il s'agit là d'une règle provenant d'un Meta-Scénario qui a validé sa "condition" et qui exécute sa partie action.

Une règle d'un scénario target "en exécution" : il s'agit d'une règle provenant d'un Target-Scénario qui a validé sa "condition" et qui exécute sa partie action.

Une règle d'un scénario target "en attente" : de manière générale, ces règles sont plus marginales. En effet, une règle "target" qui est en attente est, soit en attente d'un événement extérieur, soit en attente d'une autre règle "target". Toutefois le premier cas est plus rare, car ce sont les scénarios "meta" qui traitent ces événements en général.

De ces quatre types de règle, il est possible d'identifier plusieurs problèmes :

— Même lorsqu'il ne se passe rien, le moteur doit exécuter une série de règles en continu.
— Chaque règle en attente exécute sa méthode "dataAcquisition" à chaque itération, ce qui est un poids pour les différentes ressources.

Ces deux problèmes sont schématisés dans la figure 4.3. Cette dernière montre qu'à chaque itération une règle vérifie une condition auprès d'une ressource. Cette vérification se fait à chaque itération, même si la ressource invalide la condition. Dans un système où le moteur est exécuté en continu, ce genre de règle est malheureusement courant. Ainsi, dans une base de données reprenant une liste de salariés d'une entreprise, une règle pourrait vérifier à chaque itération si un nouvel employé est arrivé dans l'entreprise pour lui attribuer le bon statut. Dans ce genre de situation, le problème est flagrant, car la règle sera invalidée dans 99% des exécutions.

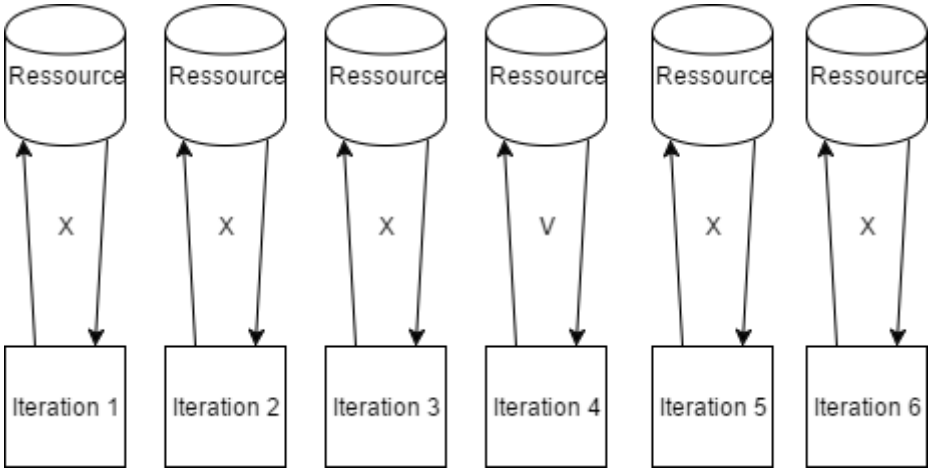

FIGURE 4.3 – Vérification continue des conditions des Meta-Scénarios

Quelles sont les solutions possibles ? Pour le premier point, il est tentant de sortir les règles en question des itérations et de ne les y placer dans une itération que lorsque survient un changement qui peut valider la condition. Pour ce faire, il va être nécessaire d'inverser le sens de la communication entre le moteur et les ressources. Si la ressource peut notifier le moteur d'un changement pertinent, il est possible de supprimer les règles en attente et de n'avoir que des règles actives. Ainsi, la ressource enverrait une notification au moteur qui générerait les règles des Meta-Scénario en conséquence, et les placerait dans la prochaine itération. Le changement est montré dans la figure 4.4, où la ressource signale son changement lors de l'itération 4, ce qui permet d'exécuter la règle en conséquence.

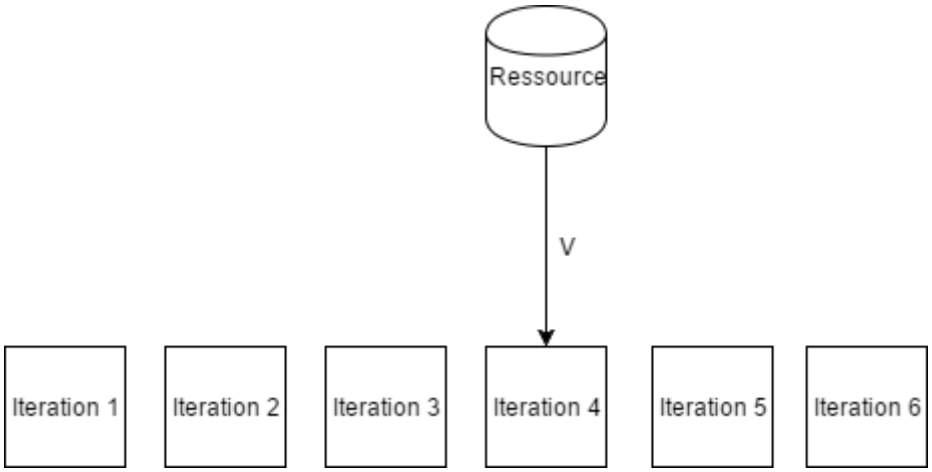

FIGURE 4.4 – Vérification événementielle des conditions des Meta-Scénarios

Toutefois, cela implique que la ressource puisse notifier le moteur. Or, l'utilisation actuelle du moteur proactif implique qu'il soit greffé sur un système existant potentiellement non modifiable. Des accès à la base de données sont autorisés, mais les logiciels en place ne sont pas toujours modifiables.

Dès lors, il est nécessaire de trouver comment le problème peut être contourné.

4.2.1 Logiciel modifiable

Le premier cas, le plus simple, est celui d'un moteur proactif destiné à modifier, gérer, interagir soit avec lui-même (le moteur fait partie de l'application finale), soit avec un logiciel que nous pouvons modifier.

Dans cette situation, il faut repérer les zones du code du programme source qui traitent un événement qui doit être géré dans le moteur, et envoyer depuis cet endroit une notification à celui-ci. Cette gestion peut se faire de plusieurs manières qui ne seront pas expliquées ici. Citons toutefois la possibilité de l'utilisation de la programmation orientée aspect qui va permettre d'intégrer ces notifications sans toucher directement au code-source lui-même.

4.2.2 Accès à une base de données

Dans le cas où le moteur a accès à une base de données, il existe plusieurs possibilités. Actuellement, comme le montre la figure 4.5, le moteur analyse les changements de la base de données en temps réel en se connectant à celle-ci et en vérifiant si des modifications ont eu lieu depuis la dernière vérification (un système de cache existe afin d'alléger cette vérification).

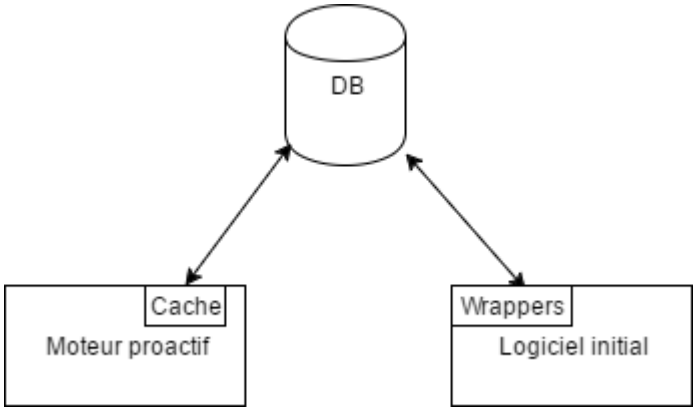

FIGURE 4.5 – La vérification des modifications sur une base de données depuis le moteur proactif

4.2.2.1 Modifier les wrappers

La solution la plus efficace reste bien entendu de modifier les accès à la base de données depuis le programme source afin de rajouter une notification au moteur proactif. Ceci rappelle évidemment le point précédent où il était possible de modifier le code source du projet initial. Plusieurs moyens sont possibles, avec différents niveaux d'impact sur le code source, mais s'il s'agit principalement de choix de designs architecturaux, le résultat reste le même. La figure 4.6 montre comment le moteur peut récupérer les changements effectués sur la base de données directement grâce aux wrappers du programme initial.

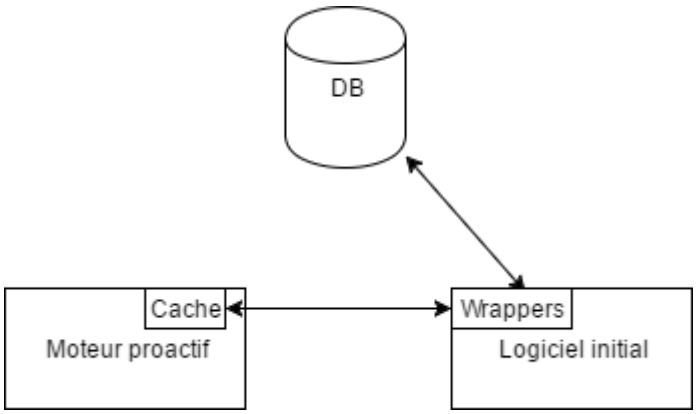

FIGURE 4.6 – Actualisation des modifications depuis les wrappers du programme initial

4.2.2.2 Modifier la base de données

Afin de détecter des changements dans une base de données en modifiant celle-ci, plusieurs méthodes sont possibles :

1. Copier les tables

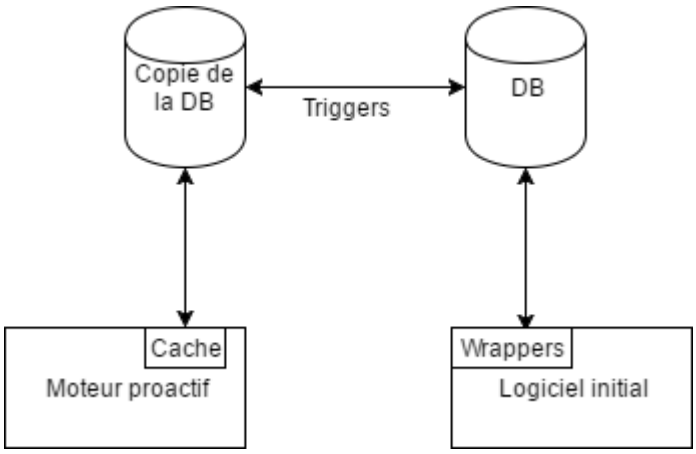

FIGURE 4.7 – Accès à une copie de la base de données

Une première possibilité est de créer un double de chaque table qui contiendrait les changements effectués sur celle-ci, comme illustré dans la figure 4.7. Grâce à des triggers bien choisis, il est ainsi possible de placer dans des tables annexes tous les changements opérés sur la table. Voici un exemple de trigger capable de récupérer

et placer dans une table annexe les changements d'une table :

```
create trigger saveUpdate
on TheTable
before update
as
insert into BeforeUpdate (Id, Data, Other)
select Id, Name, Other
from deleted ;
go
insert into AfterUpdate (Id, Data, Other)
select Id, Name, Other
from inserted ;
```

Il suffirait alors de vérifier ces tables avec le moteur, ce qui permet de limiter l'encombrement des autres tables, mais pas de la base de données elle-même. Toutefois, cette méthode n'est pas simple à mettre en place, car cela peut nécessiter beaucoup de nouvelles tables et l'écriture de beaucoup de triggers. De plus, une modification du schéma de la base de données demande de modifier les triggers et les schémas des tables doublons. De manière générale, on préfère éviter les clones [11]. En effet les clones sont toujours problématiques lorsqu'il s'agit de les mettre à jour. Dans notre situation, si le moteur proactif doit être capable de faire des modifications sur la base de données, il est préférable qu'il les exécute sur la base de données originale, de peur que l'ancienne ne soit pas à jour et que les données transférées soit erronées. Dès lors, le schéma ressemblerait plutôt à la figure 4.8

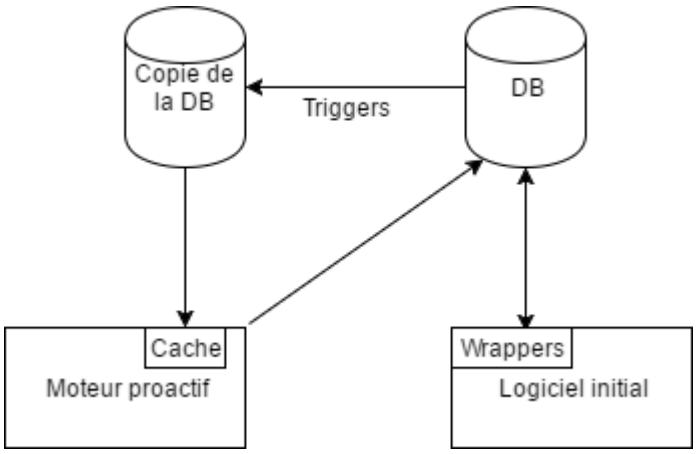

FIGURE 4.8 – Actualisation des modifications depuis les wrappers du programme initial

2. Notifier par triggers
   Une autre solution que de créer des tables supplémentaires serait de créer des triggers qui pourraient notifier le moteur d'une modification dans la table. De cette manière, plus de requête nécessaire sur les tables. On recevrait directement les données grâce à ces triggers ! L'avantage de cette méthode est évidemment d'avoir à ne pas modifier le code source, mais aussi de risquer d'altérer la base de données mise en place. Il faut toutefois pouvoir accéder à la base de données pour créer ces triggers. La figure 4.9 montre le sens des échanges de données entre le programme,

le SGBD (Système de gestion d'une base de données) et ses triggers et le moteur proactif.

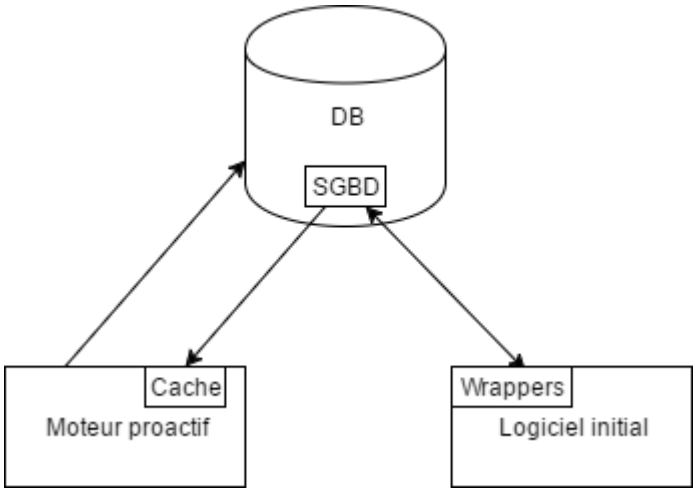

FIGURE 4.9 – Réception des modifications par le SGBD et ses triggers

3. Parser les logs
   Une autre solution encore moins invasive que les précédentes est la possibilité de parser les logs de la base de données. Si on omet les logs inintéressants, il est possible de récupérer les informations nécessaires. Cette méthode permet effectivement d'être moins invasif pour la base de données, mais demande un rafraîchissement régulier. Il n'y a donc pas de possibilité d'effectuer une notification depuis les logs vers le moteur. Ce système est illustré dans la figure 4.10.

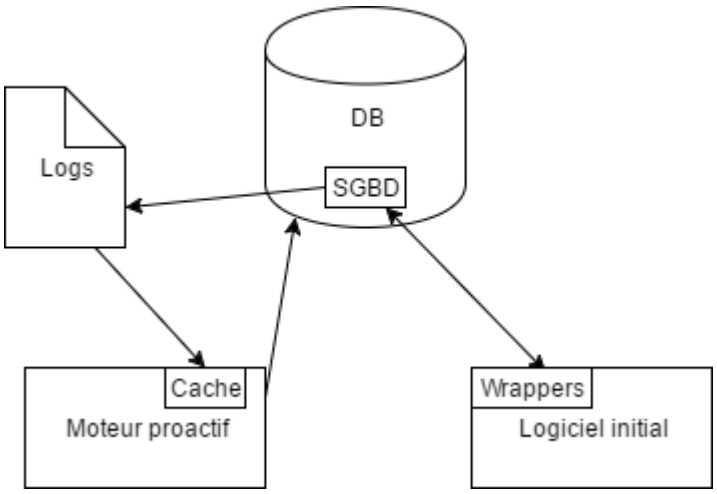

FIGURE 4.10 – Réception des modifications par le SGBD et ses triggers

4.2.3 Accès à un autre moteur

La discussion entre moteurs ne sera pas traitée dans ce mémoire, mais des pistes de solutions sont rapidement envisageables. En effet, un moteur A pourrait exécuter un

Target-Scénario pour émettre un message au moteur B qui le captera comme notification et créera un Meta-Scénario en réponse.

4.2.4 La suppression des itérations

Supprimer les itérations est un grand changement dans le fonctionnement actuel du moteur. Pourquoi cela serait-il intéressant et comment le faire ?

Que recherche-t-on en supprimant les itérations ? Tout d'abord perdre ces paramètres N et α, afin de pouvoir réagir plus vite lorsqu'un événement survient. En effet, tout événement se produisant durant le laps de temps α doit attendre l'itération suivante, de même pour un événement survenant durant une itération en cours si le paramètre N est trop petit. Enfin, couplé avec la précédente piste de solution, supprimer les règles mortes ou les rendre adaptables en fonction de leurs caractéristiques.

Tout d'abord, il faut partir de l'hypothèse que le moteur ne va effectuer du travail utile qu'en cas d'événement nécessitant ce travail. En effet, si, même sans stimuli, le moteur effectue des tâches en continu, supprimer les itérations risque d'aboutir à une "boucle infinie". En fait, il ne faut pas que le paramètre α soit utilisé pour éviter un travail continu, mais bien pour alléger le poids du moteur à la fois en interne et à la fois pour les ressources externes.

Si on revient sur le paramètre α, on peut se rendre compte qu'il était utile pour empêcher des vérifications de condition de s'exécuter 1000 fois ou plus par seconde, ce qui est inutile, mais par contre augmentait le temps d'exécution d'un scénario par le délai induit entre chaque itération.

L'idée en supprimant les itérations serait alors d'introduire un paramètre α propre à chaque règle si nécessaire, et laisser les autres scénarios (dont la condition est validée) s'exécuter au plus vite.

Des modifications sont alors possibles aux différents composants du moteur.

4.2.4.1 Une seule liste d'entrée et de sortie

Si le choix est pris de laisser les itérations de côté pour une exécution continue et événementielle du moteur, il n'est plus nécessaire d'avoir deux listes. En effet, sans itération, il n'y a plus de moment où les règles de la liste sortante peuvent être placées dans la liste entrante. La figure 4.11 rappelle le fonctionnement du paramètre α : entre deux itérations, un délai d'une durée α permet au moteur de soulager les ressources et de passer le contenu de la liste résultat dans la liste d'entrée (ici la règle 1 et 2 se répliquent).

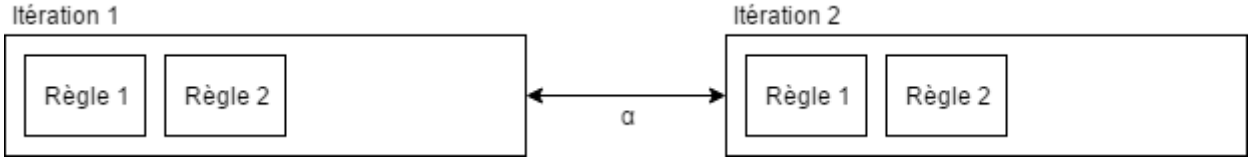

FIGURE 4.11 – Le paramètre alpha entre deux itérations

Le moteur possède donc à la place une seule liste qu'il va gérer en continu. Toutefois, il n'est pas question qu'il exécute chaque règle de la liste tant qu'il en reste, sous peine

d'en arriver à une consommation excessive du CPU. Il nous faut donc un mécanisme pour soulager le moteur ou modifier notre gestion de la liste.

Revenons sur nos différents types de règles :

1. Une règle d'un Meta-Scénario "exécutable".
   Cette règle ayant sa condition validée, elle se doit d'être traitée au plus vite. En restant dans la liste et en gardant le système actuel, ce sera bien le cas. Voyons plus avant les autres types de règle
2. Une règle d'un Target-scenario "exécutable".
   De même que pour le premier point, la règle doit être exécutée. Rien à faire de plus donc.
3. Une règle d'un Meta-scenario "en attente".
   C'est ici que se compliquent les choses. En effet, si on laisse ces règles dans la liste, le moteur va boucler indéfiniment sur les conditions de ces règles, jusqu'à ce qu'elles soient toutes validées. Malheureusement c'est un gros problème qu'il faut par-dessus tout éviter. De plus, les Meta-Scénario n'ont pas nécessairement besoin de tant de réactivité ! En effet, certains se contentent de vérifier le jour, voire le mois ou l'année actuelle ! Si on prend un scénario qui demande aux utilisateurs de changer leur mot de passe tous les six mois, il est inutile de vérifier cette condition 200 fois par seconde...
4. Une règle d'un Target-Scénario "en attente".
   Le constat est ici le même que pour les Meta-Scénario.

Il va falloir rajouter un mécanisme pour limiter l'impact de ces règles "en attente". Comme il est toujours question de garder de la réactivité sur la liste actuelle, il faut partir sur un autre principe : la liste va contenir uniquement les règles qu'il est intéressant de vérifier. Ces règles sont les règles qui doivent être revalidées. Comme il a été dit plus haut, toutes les règles "en attente" n'ont pas besoin du même taux de rafraîchissement. Le moteur n'utilisant plus d'itérations impliquant un rafraîchissement régulier de ces règles, il est possible de recréer un paramètre α, mais propre aux règles elles-mêmes et non à l'itération ! En agissant de la sorte, chaque règle pourra choisir sa fréquence d'actualisation en fonction de ses besoins. La figure 4.12 montre l'exécution des règles 1 et 2 de la figure 4.11, mais avec un délai d'exécution propre à chacune.

FIGURE 4.12 – Le paramètre alpha entre deux exécutions d'une règle.

En prenant en compte le premier principe, les règles qui se trouvent dans leur laps de temps α ne peuvent se trouver dans la liste. Il faut donc créer une seconde liste, qui va contenir les règles "en attente". Ces règles ne sont pas conservées dans la liste actuelle, sous peine de retrouver le problème initial : le moteur va devoir constamment vérifier si le temps α est écoulé ou non. Certes, cela consomme moins de temps de calcul que certaines conditions, mais le moteur reste actif. La solution est acceptable, mais il est encore possible d'améliorer la situation actuelle.

4.2.4.2 Une liste de règles en attente

Le moteur va alors posséder une nouvelle liste parallèle à la première, qui va contenir toutes les règles qui ne doivent pas être vérifiées à l'instant présent. Il s'agit bien ici d'instance de règle, et non de règle en terme de classe. Imaginons que nous plaçons dans cette liste trois règles A, B et C qui, chacune, ont respectivement un paramètre α équivalent à 1000, 5000 et 900000 ms (figure 4.13).

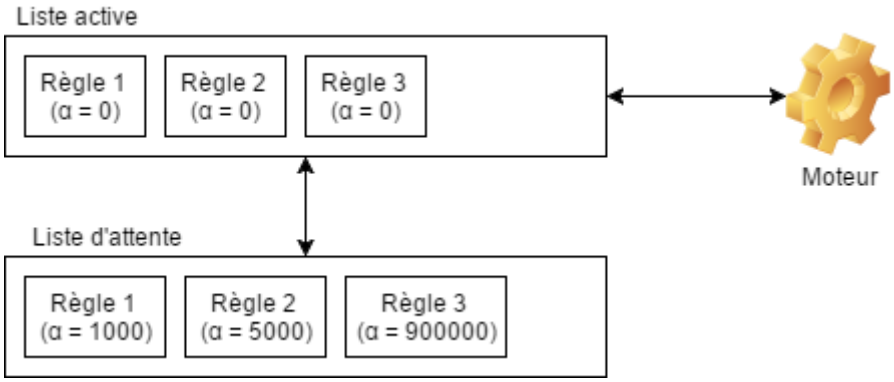

FIGURE 4.13 – Liste d'attente pour les règles avec un alpha non nul. Le moteur récupère les règles de la liste active, tandis que les règles de la liste d'attente ne sont pas exécutées par le moteur.

Il faut donc que, dans 1000 ms, la première règle soit déplacée dans la liste des règles actives. Pour gérer cette transition, deux moyens sont possibles : la méthode synchrone ou asynchrone.

En version synchrone, il faut une règle spécifique qui reste dans la liste active et qui a pour condition l'évaluation du plus petit temps α des règles de la liste d'attente. Lorsque la condition est validée, le moteur peut aller rechercher cette règle dans l'autre queue et la placer dans la liste actuelle. Il ne restera plus qu'à générer une nouvelle règle avec une nouvelle évaluation du plus petit temps α. On en revient toujours à l'évaluation du temps dans la liste active, avec la différence qu'une seule règle est vérifiée de cette manière dans la liste. Il y a donc un intérêt (faible, mais présent tout de même) si le nombre de règles en attente est conséquent. Qui plus est, cela permet d'externaliser les règles en attente de la liste active et d'opérer une gestion plus approfondie de la mémoire pour ces règles (choix de l'utilisation d'une liste ou d'une map, tri en fonction des paramètres α, sous-listes, références entre règles, etc...). Toutefois, cette solution ne sera pas conservée et ne sera plus abordée par la suite, car l'objectif est d'enlever au maximum les règles mortes.

La version asynchrone, qui sera réutilisée par la suite, présente l'avantage de s'auto-gérer. L'idée est d'avoir un thread dédié à la gestion de la liste d'attente. Lorsqu'il faut attendre l'écoulement du temps du plus petit paramètre α, le thread est endormi, et au temps donné, il se réveille afin de placer ladite règle dans la liste active. Si le thread est endormi au moment où une règle est placée dans la liste d'attente, le thread se réveille et se met à jour. Ainsi, plus de trace des règles en attente dans la liste active !

Voyons une implémentation possible de cette classe :

```
public class TimedQueue implements Runnable {

  private ActiveQueue activeQueue;
  private Map<AbstractRule, Long> queue;

  private boolean running;
  private long minTimeToWait;

  public TimedQueue(ActiveQueue activeQueue) {
    this.activeQueue = activeQueue;
    queue = new HashMap<AbstractRule, Long>();
  }

  public synchronized void add(AbstractRule rule) {
    queue.put(rule, System.currentTimeMillis());
    notify();
  }

  @Override
  public void run() {
    running = true;
    while (isRunning()) {
      long now = System.currentTimeMillis();
      minTimeToWait = 10000;
      Iterator<Entry<AbstractRule, Long>> it =
            queue.entrySet().iterator();
      while (it.hasNext()) {
        Entry<AbstractRule, Long> pair = it.next();
        if (pair.getValue()
          + pair.getKey().getTimeToWait() <= now) {
          pair.getKey().setTimeToWait(0);
          activeQueue.add(pair.getKey());
          it.remove();
        } else {
          long toWait = pair.getValue() +
            pair.getKey().getTimeToWait() - now;
          if (toWait < minTimeToWait)
            minTimeToWait = toWait;
        }
      }
      waitForRuleToMove(minTimeToWait);
    }
  }

  private synchronized void
          waitForRuleToMove(long minTimeToWait) {
    try {
      wait(minTimeToWait);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  private synchronized boolean isRunning() {
    return running;
  }

  public synchronized void stop() {
```

```
59        running = false;
60        notify();
61      }
62    }
```

Le début du code possède un lien vers la liste active :

```
1     private ActiveQueue activeQueue;
```

Cette liste non présentée en code ici est bien entendu thread-safe. Cette liste étant gérée par un thread, il est donc nécessaire de la démarrer et de la stopper, d'où la méthode run et stop.

La méthode run va effectuer une vérification du temps restant avant déplacement vers la liste active de chaque règle de la liste, et si le temps est écoulé, la déplacer effectivement. Lorsque cette vérification est terminée, le thread va se mettre en attente pour une durée équivalant au plus petit temps rencontré dans les différentes règles. Dans cet exemple, cette durée ne peut être plus grande que 10 secondes.

Grâce à la méthode add, il est possible d'ajouter une règle dans cette liste (méthode thread-safe), ce qui a pour effet de réveiller instantanément le thread qui va réeffectuer une vérification du timing. Le choix a été pris de revérifier toutes les règles, mais il est tout à fait cohérent de ne vérifier que la règle entrante. Cette décision vient du possible impact de la nouvelle règle sur d'autres règles.

4.2.4.3 Une gestion économe de la liste principale

L'implémentation asynchrone de la liste d'attente permet d'analyser un autre problème qui a été évoqué plus tôt concernant le thread principal du moteur. Celui-ci est censé exécuter toute règle présente dans la liste principale. Or, si aucune règle n'est présente, il va vérifier la présence de règle dans la liste en continu, et ainsi boucler jusqu'à ce qu'une nouvelle règle arrive dans la liste. Or, ce comportement est problématique car il va consommer une grande quantité de CPU, et potentiellement bloquer le bon fonctionnement du programme, voire même de la machine.

Une solution possible pour alléger l'impact de ce problème est d'intégrer un sleep avec un délai minuscule entre chaque vérification de la liste (environ 10ms). Toutefois, malgré la petitesse du temps utilisé, cela revient à mettre en place des itérations séparées par un tout petit α.

Afin d'éviter cela, il faut repartir de la solution du point précédent et implémenter un wait du thread principal du moteur qui s'active lorsqu'il n'y a plus de règle dans la queue et qui se réveille lorsqu'une nouvelle règle y est ajoutée avec le mécanisme wait/notify, par exemple.

4.3 Exécuter les règles en asynchrone

Il reste actuellement un problème sans solution : comment conserver de la réactivité lorsqu'une règle prend beaucoup de temps à s'exécuter ?

De la manière dont le moteur a été développé, les règles doivent être les plus petites possibles. Toutefois, cela n'est pas toujours faisable. Il peut s'agir de calculs complexes ou de connexions vers l'extérieur qui prennent du temps. Et, tant qu'une règle de ce genre s'exécute, le reste du moteur est dans l'incapacité de répondre à un autre événement.

Il parait évident qu'il va falloir jouer avec le parallélisme, mais comment ? En effet, plusieurs méthodes sont possibles.

4.3.1 Un cœur synchrone avec annexes asynchrones

La première solution est de conserver le moteur actuel, tout en garantissant qu'une règle soit courte à exécuter. Pour cela, il est possible de donner au programmeur un moyen simple de lancer des threads parallèles effectuant les tâches lourdes. Cette technique est illustrée dans la figure 4.14.

Cette solution peut fonctionner, mais uniquement si le programmeur de règles pense bien à externaliser les parties plus lourdes de son code. C'est d'ailleurs une difficulté de cette pratique : quand peut-on dire que le code est long à exécuter, que faut-il externaliser ? De plus, le code n'est pas nécessairement toujours apte à supporter le multithreading. Si l'externalisation de la complexité se fait mal, cela pourra provoquer de temps à autre des erreurs de concurrence, et tout le monde sait que ce sont les pires erreurs à traquer et corriger ! Qui plus est, cette méthode empêche de garder le principe de l'entièreté d'une règle : si une règle se termine, c'est que l'on considère que son travail est terminé. Dans ce cas de figure-ci, on ne sait pas quand le code a fini de s'exécuter, à moins de recréer une règle en sortie de processus.

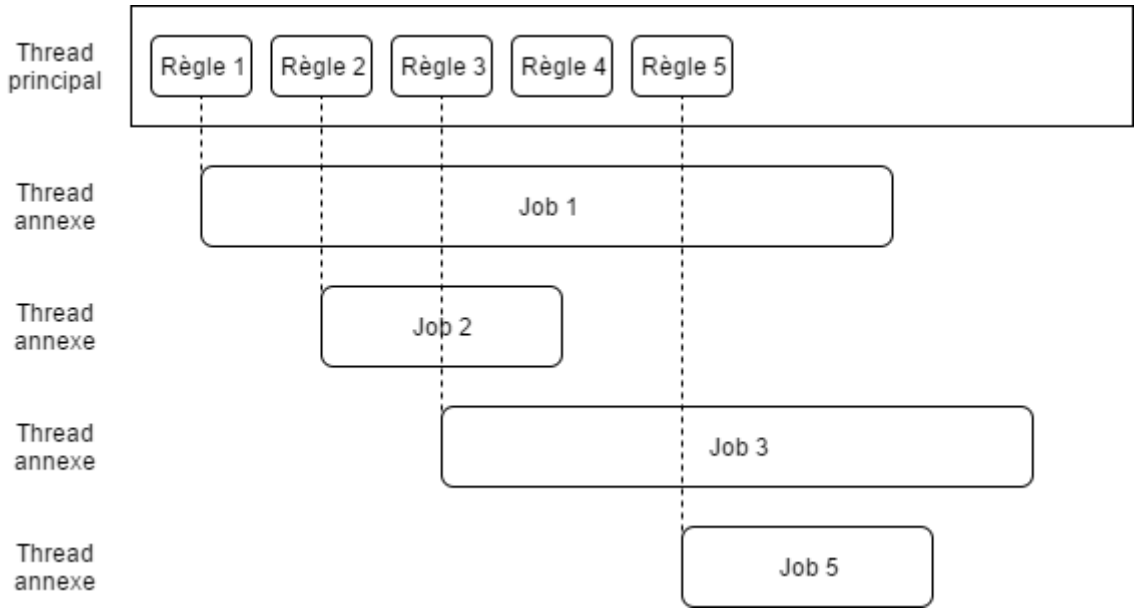

FIGURE 4.14 – Exécution de cinq règles, dont quatre nécessitent l'exécution d'un travail parallèle.

4.3.2 Un cœur asynchrone avec annexes synchrones

Une autre solution est d'exécuter les règles directement dans des threads différents. Ainsi, en rendant la concurrence obligatoire, on assure une certaine réactivité grâce à une grosse réduction des blocages entre règles. Une première vision est montrée dans la figure 4.15.

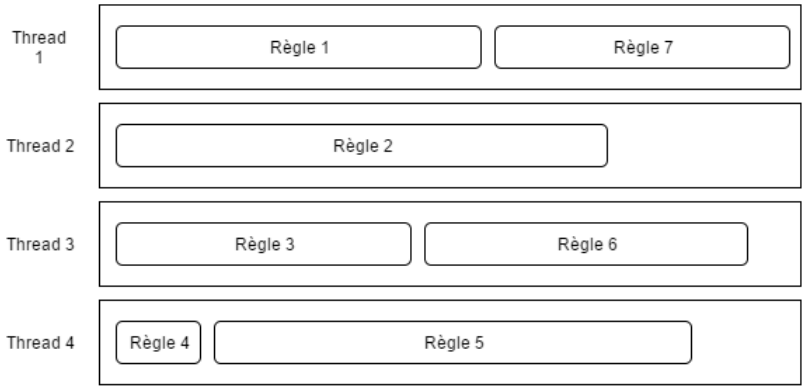

FIGURE 4.15 – Exécution de sept règles sur quatre threads.

La suite du mémoire se centrera sur cette solution, et tentera de résoudre les problèmes inévitables qui vont se présenter lors de l'implémentation. En particulier sur la gestion des ressources.

Chapitre 5

Architecture générale du moteur proactif asynchrone

5.1 Vue d'ensemble de la version asynchrone

La présent chapitre présente une version asynchrone du moteur proactif cité aux chapitres précédents basé sur un système de règles. La plupart des pistes explorées au chapitre 4 sont implémentées ici.

Le système essaiera de passer le plus possible sur un système événementiel, et là où ce n'est pas possible, tout sera fait pour alléger au maximum le poids des règles "en attente".

L'exécution des règles sera asynchrone, chaque règle étant exécutée dans un thread parallèle.

5.2 Architecture générale

5.2.1 Des règles

Évidemment, le moteur n'est rien sans règles. Comme dans l'ancien moteur, une règle est une partie d'un scénario, qui peut être un Meta ou un Target-Scenario. Il n'y a qu'une classe pour les Meta ou Target-Scenarios, c'est le comportement de la règle qui va déterminer son rôle. Toute règle instancie donc une classe générique pour les règles, que nous appellerons AbstractRule.

5.2.2 Un thread principal

Le thread principal du moteur va être chargé de sélectionner les règles à exécuter dans la liste de règles dédiée à cet effet. Actuellement, c'est la seule tâche dont il est chargé, les autres tâches possibles seront abordées plus loin dans ce chapitre.

5.2.3 Un pool de threads

Afin de rendre la concurrence du moteur paramétrable, les règles seront exécutées dans un pool de threads qui contiendra un nombre prédéfini de threads, dédiés à l'exécution de règle. La figure 5.1 montre le transfert des règles de la liste active vers le pool de threads.

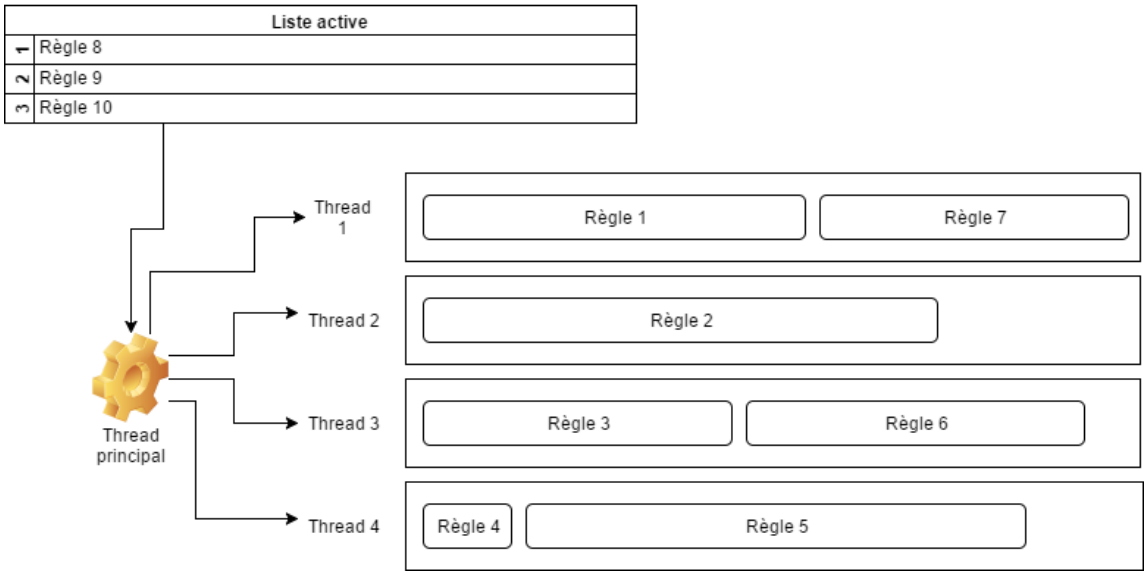

FIGURE 5.1 – Exécution de règles provenant de la liste active dans le pool de threads.

5.2.4 Des ressources

Enfin, il est nécessaire de conserver les différentes ressources précédemment utilisées : classes internes, connexions extérieures, etc. Désormais, toutes les ressources, quelles qu'elles soient, devront implémenter la classe AbstractRessource. L'intérêt est dans la gestion de la concurrence.

5.3 Gestion des ressources

Dans un environnement concurrentiel, la principale difficulté réside dans la gestion des ressources partagées. Dès lors, l'objectif est de placer dans le moteur un support le plus simple possible pour gérer cette concurrence tout en laissant la liberté au programmeur de faire ce qu'il désire. Un outil flexible et puissant, tel est l'objectif à atteindre.

Partons d'un exemple simple :

Une règle X utilise une classe A pour son exécution. Malheureusement, une autre règle, Y, l'utilise aussi. Dans la version synchrone du moteur, il n'y aurait eu aucun problème à ce genre de cas, chaque règle étant exécutée l'une après l'autre. Mais ici, notre objectif est d'exécuter plusieurs règles en même temps. Cet accès doit donc être contrôlé, sous peine d'avoir des erreurs de concurrence, ce qui peut amener à divers soucis, tels que des problèmes de consistance des données et donc en général de crash du système ou des comportements anormaux.

Dans cette situation, il faut clairement mettre en place un mécanisme de verrou sur les accès aux ressources partagées. Le projet étant développé en Java, une première solution serait de rendre les ressources thread-safe en assurant que les méthodes d'accès aux ressources soient contrôlées par le mot-clé "synchronized".

Cette première solution rendrait le code stable car nous n'aurions plus d'erreurs de concurrence. Toutefois cette gestion peut amener à des erreurs de cohérence. Le cas suivant montre un exemple d'erreur qui peut survenir :

Un événement A se produit. Cet événement correspond à une modification dans une ressource, disons B. Lorsque cette ressource est modifiée et donc lorsque l'événement A se produit, deux scénarios s'exécutent : le scénario X vérifie que le reste de la ressource est toujours intègre et modifie si nécessaire le changement de données. Le scénario Y récupère le changement, le compare à d'autres ressources et propage ce changement à ces ressources.

Enfin, cette situation peut être déclinée en deux cas. Dans le premier, tout se passe correctement, mais dans le deuxième un problème typique de la concurrence survient :

1^{er} cas

- La ressource B est modifiée : la variable C passe de 5 à 10.
- X vérifie que la ressource soit bien intègre.
- X a terminé sa vérification et estime qu'il faut changer la variable C à 3 et non à 10.
- Y récupère la variable C, actuellement à 3.
- Y propage ce changement (C=3) à d'autres ressources

2^{eme} cas

- La ressource B est modifiée : la variable C passe de 5 à 10.
- X vérifie que la ressource soit bien intègre.
- Y récupère la variable C, actuellement à 5.
- X a terminé sa vérification et estime qu'il faut changer la variable C à 3 et non à 10.
- Y propage ce changement (C=5) à d'autres ressources

Note : Il faut considérer que l'événement n'est pas directement généré par la modification de la ressource mais provient d'une source extérieure. Dès lors, lorsque le scénario X modifie C, aucun événement n'est généré. Dans ce cas-ci, une solution pourrait être de générer un événement à ce moment, mais l'objectif ici est de régler le problème de la concurrence.

Pour simplifier l'utilisation des règles et laisser le moins possible au programmeur de règles la gestion de la concurrence, il faut partir du principe qu'une règle gardera son accès à une ressource tant qu'elle est exécutée. Cela fait sens si les règles sont effectivement développées comme ayant un objectif unique. Une règle fait une tâche. Si un scénario nécessite plusieurs actions, alors il est possible d'enchainer les règles entre elles. Si ce principe est respecté, alors l'accès à une ressource pour une règle est effectivement conservé lors de toute son exécution. En faisant cela, on empêche toute autre règle de compromettre le bon fonctionnement d'une règle.

Pour ce faire, il convient dès lors de gérer cet accès aux ressources, par exemple par des verrous. Les sections suivantes montrent d'abord comment mettre en place différentes ressources, et enfin comment gérer l'accès à ces ressources depuis les règles.

5.3.1 Accès avancés

En partant du principe qu'une règle accède seule à une ressource durant son exécution, le risque de perdre une bonne partie des bénéfices apportés par la concurrence apparaît. Il convient donc d'affiner cette restriction des accès à une ressource.

En effet, dans le cas où deux règles veulent récupérer une valeur d'une ressource mais pas la modifier, alors ces deux règles pourraient accéder en même temps à la ressource. Par contre, ce ne sera pas le cas des règles qui veulent modifier la valeur en question. Celles-là ne pourront s'exécuter pendant aucune autre règle.

Ce qui se remarque rapidement, c'est que la gestion des accès va varier en fonction de la règle. Il va donc falloir que le moteur, au début de l'exécution de la règle, vérifie la disponibilité des ressources qui seront utilisées.

5 types d'accès à une ressource ont alors été définis :

1. Aucune ressource
2. Aucun verrou
3. Verrou en lecture
4. Verrou en écriture
5. Verrou en écriture, accès forcé

Les verrous 1 et 5 ne sont pas disponibles au programmeur de règles. En effet, ces verrous sont utilisés par le moteur de manière cachée (cela lui permet d'avoir des valeurs minimum et maximum à comparer, et de forcer l'accès à une ressource si cette dernière est bloquée pour une obscure raison).

Le deuxième verrou est en réalité une absence de verrou. Il peut être utilisé pour vérifier la présence d'une ressource, mais pas y accéder.

Enfin les quatrième et cinquième verrous sont les plus intuitifs : pour le premier, on souhaite lire des données de la ressource et pour le second y écrire des données.

5.3.2 Implémentation des règles

Maintenant que le type de moteur et les verrous ont été retravaillés, le code suivant présente une première ébauche de code pour la classe mère des règles :

```
1   public abstract class AbstractRule
2         implements Runnable, Cloneable {
3     protected int id;
4     private long timeToWait;
5     protected GenericAsynchronousProactiveEngine engine;
6
7     private Map<String, Lock> locks;
8
9     private RuleType type = RuleType.GENERIC;
10
11    public AbstractRule
12          (GenericAsynchronousProactiveEngine engine) {
13      locks = new HashMap<String, Lock>();
14      this.engine = engine;
15    }
16
17    protected AbstractResource
18          addLock(String resource, LockType lockType) {
19      AbstractResource iResource = null;
20      if (ResourceMgr.getInstance().
21                  getResource(resource) != null)
22        iResource = ResourceMgr.
23                getInstance().getResource(resource);
24      else
25        Global_Vars.logger.warning
26        ("Be careful! Resource " + resource +
27                    " not found!");
28      Lock lock = new Lock(iResource, lockType);
29      locks.put(resource, lock);
```

```
30        return lock.getResource();
31      }
32
33      public abstract void register();
34
35      protected abstract void dataAcquisition();
36
37      protected abstract boolean activationGuards();
38
39      protected abstract boolean conditions();
40
41      protected abstract void actions();
42
43      protected abstract boolean rulesGeneration();
44
45      @Override
46      public abstract String toString();
47
48      public void createRule(final AbstractRule rule) {
49        engine.addRule(rule);
50      }
51
52      public void run() {
53        int executionId = this.executionId;
54        dataAcquisition();
55        if (activationGuards()) {
56          if (conditions()) {
57            actions();
58          }
59          rulesGeneration();
60        } else {
61          setTimeToWait(1000);
62          createRule(this);
63        }
64        ResourceMgr.getInstance().freeLocks(this);
65        QueueMgr.getInstance()..addFinishedRule(this);
66      }
67
68      protected final long getId() {
69        return this.id;
70      }
71
72      protected final void setId(int i) {
73        this.id = i;
74      }
75
76      public final RuleType getType() {
77        return this.type;
78      }
79
80      public final void setType(RuleType type) {
81        this.type = type;
82      }
83
84      public long getTimeToWait() {
85        return timeToWait;
86      }
87
```

```
88      public void setTimeToWait(int i) {
89        timeToWait = i;
90      }
91
92      public Map<String, Lock> getLocks() {
93        return locks;
94      }
95    }
```

Ce code possède une bonne partie des différents éléments abordés dans les pages précédentes :

— La règle implémente l'interface Runnable.
En effet, maintenant la règle doit être capable d'être lancée dans un thread dédié. Pour ce faire, elle doit pouvoir s'autogérer. Ainsi, la méthode "run" décrit le comportement d'une règle en fonction des différents résultats des gardes :

```
1       public void run() {
2         int executionId = this.executionId;
3         dataAcquisition();
4         if (activationGuards()) {
5           if (conditions()) {
6             actions();
7           }
8           rulesGeneration();
9         } else {
10          setTimeToWait(1000);
11          createRule(this);
12        }
13        ResourceMgr.getInstance().freeLocks(this);
14        QueueMgr.getInstance().addFinishedRule(this);
15      }
```

Les explications des deux dernières lignes se trouvent à la section 6.2.1.

— La règle possède une variable indiquant le délai avant lequel la règle doit être exécutée.
La variable timeToWait permet en effet de postposer l'exécution d'une règle. Ceci a deux effets : le programmeur de règles ne doit pas gérer les différentes queues actives et non actives, et le retard avant une exécution est très simple à paramétrer ou changer. Grâce à cette information, le moteur est capable de gérer les deux types de queues de manière cachée. Ainsi, lorsqu'une règle est ajoutée dans le système, elle est automatiquement ajoutée à la bonne liste. Ceci apparaît d'ailleurs dans la méthode "createRule" :

```
1       public void createRule(final AbstractRule rule) {
2         engine.addRule(rule);
3       }
```

Cette méthode est utilisée dans la fonction ruleGeneration. Elle permet de rajouter la règle donnée en paramètre dans le système. C'est donc le temps indiqué dans "rule.timeToWait" qui donne l'information au moteur du traitement à prodiguer à la règle.

— La règle possède une liste de verrous

Enfin, et c'est la grosse partie concernant la concurrence, une règle possède la liste des verrous qui seront nécessaires à son exécution dans la variable "locks".
Il s'agit d'une liste contenant, pour chaque ressource nécessaire, le niveau de verrou dont la règle aura besoin pour s'exécuter. Il convient donc au programmeur de règles de lister les ressources utiles à la règle dans la méthode "register".
Cette méthode a deux utilités : remplir la liste des verrous utilisés par la règle, mais aussi récupérer ces fameuses ressources. En effet, cette méthode se remplit comme suit :

```
1       @Override
2       public void register() {
3         myResource = (ResourceClasse)
4             addLock("ResourceName", LockType.WRITE);
5       }
```

Après l'appel à register, myResource, qui est une variable membre de la règle, contient la ressource demandée (ici nous voulons un verrou en écriture (WRITE)).

5.3.3 Gestion des ressources

Auparavant, une ressource n'était pas identifiée comme telle. Entendons par là que si on voulait, par exemple, accéder à une classe, on y accédait, tout simplement. Dans une version asynchrone où tous les accès à des ressources doivent être contrôlés, il est nécessaire de lister et répertorier chaque ressource. Ainsi, le moteur possède une liste de ressources qu'il initialise au démarrage et à laquelle chaque règle peut accéder en demandant un verrou particulier.

Toutefois, lister et demander les verrous pour chaque règle et ressource peut être source d'erreurs. En effet, on n'est jamais à l'abri d'une modification de la règle ou de la ressource qui amènerait à changer, ajouter ou supprimer un verrou de la liste. Or cette dernière modification peut facilement être oubliée par le programmeur, et, si c'est le cas, on en revient à la même situation qu'initialement...

Pour empêcher cela, les accès aux ressources seront plus contrôlés. L'idée va être de bloquer l'appel à une méthode d'une ressource si la règle n'a pas demandé le verrou au préalable. Il va donc falloir indiquer pour chaque méthode de la ressource, le niveau de verrou nécessaire à l'appel de cette méthode. On peut, soit le spécifier dans la méthode elle-même, soit utiliser une programmation orientée aspect pour externaliser cette vérification du code lui-même. La figure 5.2 montre l'exécution de règles accédant à des ressources. On peut remarquer la règle 6 qui a été mise en attente que la règle numéro 2 achève sa lecture sur la ressource 2.

La section suivantes montre comment gérer cet accès en fonction du type de ressource.

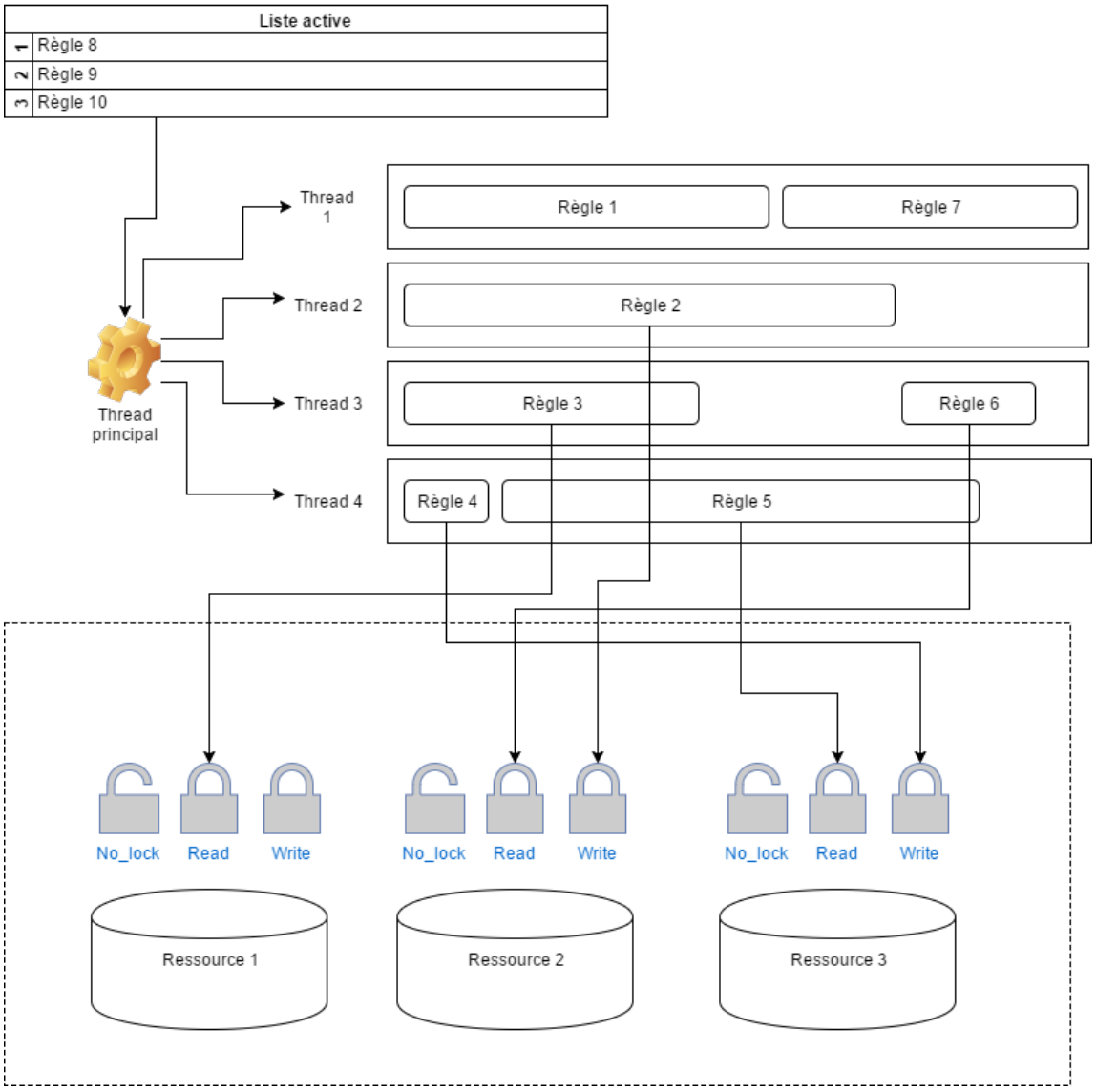

FIGURE 5.2 – Exécution de règles accédant à différentes ressources

5.3.4 Types de ressources

— Classe interne au moteur
Imaginons qu'une règle veuille accéder à une ressource interne au programme. Prenons, par exemple, une simple classe comme celle-ci : ce premier cas montre l'accès d'une règle à une ressource interne au programme. Voici un exemple de ressource interne :

```
1   public class ExampleResource {
2
3     private int internValue;
4
5     public ExampleResource() {
6       internValue = 5;
7     }
8
9     public void setInternValue(int value) {
10      internValue = value;
11    }
12
13    public int getInternValue(int value) {
14      return internValue;
15    }
16  }
```

Rendre thread-safe cette classe aurait consisté à ajouter le mot-clé synchronized devant les deux accesseurs, "setInternValue" et "getInternValue". Mais comme il a été vu plus tôt, il est préférable d'utiliser des verrous par règle. Il convient donc de bloquer l'accès aux règles qui n'ont pas le bon verrou :

```
1   public class ExampleResource {
2     private int internValue;
3
4     public ExampleResource() {
5       internValue = 5;
6     }
7
8     public void setInternValue(int value) {
9       if (hasRequiredLock(caller, LockType.WRITE))
10        internValue = value;
11      else
12        throw new BadAccessException();
13    }
14
15    public int getInternValue(int value)
16            throws BadAccessException {
17      if (hasRequiredLock(caller, LockType.READ))
18        return internValue;
19      else
20        throw new BadAccessException();
21    }
22  }
```

Avec ce système, une règle qui ne pourrait pas utiliser une méthode voulue (par manque de verrou adéquat) verrait une exception se générer. Pas d'erreurs de concurrence donc et une erreur facile à détecter pour le programmeur de règles. Il connaît facilement la source et le moment où l'erreur se produit.
— Base de données

Dans le cas d'une base de données, la ressource va être un peu différente. En effet, il existe plusieurs possibilités :

La première consiste à ouvrir une connexion au démarrage du moteur, et implémenter un wrapper avec le format de la précédente classe, où chaque méthode est contrôlée. Mais par rapport à quoi ? De manière générale les accès à la classe "Connection" de JDBC (pour prendre un exemple) est thread-safe. Donc on pourrait très bien n'utiliser aucun verrou sur les méthodes de la classe. Dans cette solution, les verrous sont alors d'ordre sémantique : on ne veut pas de modification sur la base de données durant l'exécution de la règle. Toutefois, cela ne garantit pas qu'un autre programme ne va rien modifier, et d'autres mécanismes de verrous de table existent en SQL. Au final, les verrous ne sont ici pas très utiles si la librairie de connexion à la DB est thread-safe. Dès lors un autre problème se pose : si la librairie en question est utilisée par une autre règle qui effectue des accès à la base de données, l'exécution est mise en attente de la fin du processus précédent.

Une deuxième solution consiste à créer une ressource à la volée, personnalisée à la règle. Cette ressource possède sa propre connexion. Pas besoin de verrous, et on a la garantie qu'aucune autre règle ne va accéder à la ressource en question. Mais se présente alors un nouveau problème : si beaucoup de règles s'exécutent, et comme préconisé, les règles sont les plus petites possibles, on se retrouve à ouvrir des connexions des dizaines de fois par seconde, ce qui est probablement inutile et va aussi impacter les performances.

La troisième solution est alors de proposer un pool de wrappers avec chacun leur connexion, et lorsqu'une règle demande la ressource en question, il reçoit un wrapper libre à ce moment. Cette solution permet d'éviter tous les problèmes susmentionnés : pas besoin de verrous particuliers et une connexion personnalisée.

— Connexion extérieure
De manière générale, pour toute connexion extérieure, le système sera proche du mécanisme utilisé pour les bases de données, avec un minimum d'adaptation.

5.4 Exécution des règles

Pour revenir sur la manière dont sont exécutées les règles, la liste active est maintenant composée de règles qui doivent se faire exécuter. Si cette liste n'est pas vide, le moteur va sélectionner la première règle exécutable et l'envoyer au pool de thread qui va lui réserver un thread et exécuter cette règle dans ce thread. Dans cette opération, plusieurs éléments sont à mettre en avant :

Tout d'abord, qu'est-ce qu'une règle "exécutable" ? Il ne s'agit pas d'une règle dont les conditions sont remplies, puisque la vérification des conditions se fait lors de son exécution. Il s'agit en fait d'une vérification de la disponibilité des ressources requises par la règle. Ce qui revient à vérifier, pour chaque verrou :

— S'il s'agit d'un verrou de présence, on vérifie que le moteur possède bien la ressource.
— S'il s'agit d'un verrou de lecture, on vérifie qu'aucune règle ne possède de verrou en écriture sur la ressource pour le moment.
— S'il s'agit d'un verrou d'écriture, on vérifie qu'aucune règle ne possède de verrou en lecture et écriture sur la ressource pour le moment.

Si toutes ces conditions sont remplies, on donne les verrous demandés à la ressource et on lance son exécution. Afin de savoir quelle règle a actuellement quel verrou sur une ressource, il y a deux possibilités : d'une part, la règle est en exécution, et donc la liste des

verrous qu'elle déclare sont des verrous qui lui ont été attribués, d'autre part la ressource elle-même possède une liste des règles qui ont actuellement un verrou sur elle.

5.5 Vue d'ensemble

Dans ce chapitre, nous avons mis au point une série de mécanismes qui ont mené à une version asynchrone du moteur proactif. Nous avons donc supprimé les itérations pour rendre l'exécution du moteur continue. Pour limiter le travail inutile, nous avons ajouté une liste de règles actives et une liste de règles en attente. Chaque règle possède son propre paramètre α et est capable de s'exécuter dans un thread dédié.

Dans un deuxième temps, nous avons travaillé les accès aux ressources par les règles. Nous avons fait en sorte que chaque ressource ait ses propres verrous. Chaque règle doit posséder ceux-ci pour pouvoir y accéder. Ainsi on garantit une exécution thread safe et sémantiquement valide.

Chapitre 6

Améliorations de l'architecture du moteur proactif asynchrone

6.1 Introduction

Le moteur tel qu'il a été développé permet une exécution simple d'une série de règles de manière asynchrone. Toutefois, cette exécution est peu contrôlée, et des améliorations sont évidement possibles, voire indispensables. Ce chapitre présente deux mécanismes importants qui augmentent considérablement l'efficacité du moteur.

6.2 Permettre l'ordonnancement

Le système est maintenant capable de lancer les règles de manière concurrente. Plus les règles et les ressources seront développées de manière événementielle comme expliqué au point 4.2, plus le moteur pourra être performant car débarrassé du maximum de travail inutile. Il reste toutefois à analyser l'intérêt de l'ordonnancement dans la version asynchrone du moteur.

Dans la version asynchrone, l'ordonnancement peut participer à améliorer la réactivité dans deux cas. Dans les deux cas, il s'agit de permettre aux règles plus importantes de s'exécuter avant d'autres règles qui les en empêcheraient. Mais pourquoi cela arriverait-il dans la version asynchrone ?

La première possibilité est simple : le moteur possède un pool de threads. Si ce pool n'est pas configuré comme acceptant un nombre infini de threads, il est possible qu'à un moment t, tous les threads soient occupés, auquel cas la règle attendra son tour... L'ordonnancement peut aider à résoudre ce problème, mais nous serions tenté de répondre que, si le système se retrouve effectivement bloqué pour cette raison, il faudrait probablement augmenter le nombre de threads disponibles, car cela montre que le système est débordé par les tâches à effectuer.

La deuxième possibilité est plus problématique et justement liée à la version asynchrone du système. Il peut arriver que dans certains cas, si une ressource est trop sollicitée, certaines règles demandant des accès plus restrictifs ne reçoivent jamais cet accès. Illustrons rapidement par un exemple. La figure 6.1 montre une situation où les règles 1 et 3 se répliquent sur un pool de 2 threads. Ces deux règles utilisent un accès en lecture à la ressource, ce qui empêche la règle 4 d'être exécutée, car elle nécessite un accès total à la ressource.

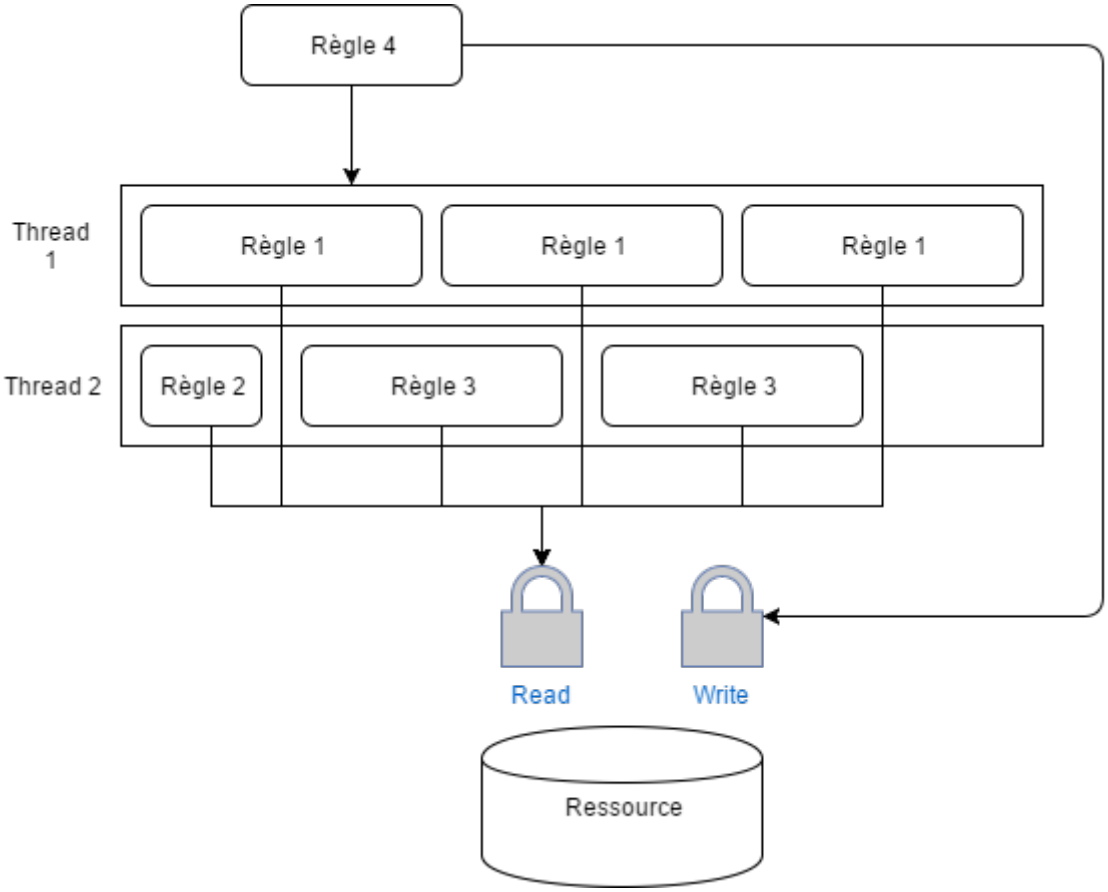

FIGURE 6.1 – La règle 4 ne peut pas être exécutée car les règles 1 et 3 bloquent l'accès en lecture en continu.

Voici deux méthodes permettant de résoudre ce problème.

6.2.1 Ordonnancement par verrou

La première solution est une méthode générique détachée de la finalité du programme et des règles qui le composent. L'objectif est d'ajouter une priorité aux règles en fonction du verrou qu'elles demandent et de classifier les verrous par ordre d'importance. Ainsi un verrou WRITE sera toujours prioritaire à un verrou READ.

(FORCE_WRITE >) WRITE > READ > NO_LOCK (> INEXISTANT)

Exemple :

Contexte : différentes règles essaient d'accéder à la ressource A dans un pool de deux threads. Les règles en READ_X sont des règles demandant un accès en lecture à la ressource, et les règles en WRITE_X sont des règles demandant un accès en écriture à cette même ressource.
Liste d'événements :

- 0ms : READ_1 commence son exécution (durée : 200 ms)
- 100ms : READ_2 commence son exécution (durée : 200 ms)
- 150ms : WRITE_1 est arrivé dans la file mais ne peut pas s'exécuter car la ressource est actuellement lue par deux règles. Toutefois, l'accès en lecture est maintenant bloqué.
- 200ms : READ_1 termine son exécution.
- 250 ms : READ_3 voudrait commencer son exécution, mais doit attendre car une règle WRITE a demandé un accès à la ressource.
- 300ms : READ_2 termine son exécution et WRITE_1 peut démarrer la sienne (pour 100ms).
- 500ms : WRITE_1 termine son exécution et READ_3 démarre la sienne.

Cette solution fonctionne, mais uniquement dans le cas où le nombre d'accès en écriture est rare et sporadique. Dans le cas où les règles WRITE sont fréquentes, il y a un risque pour que les règles en lecture n'accèdent jamais à la ressource. En effet, il s'agit d'une manière générique d'ordonnancer, et peut-être un peu trop simple. La figure 6.2 illustre la ligne du temps citée ci-dessus.

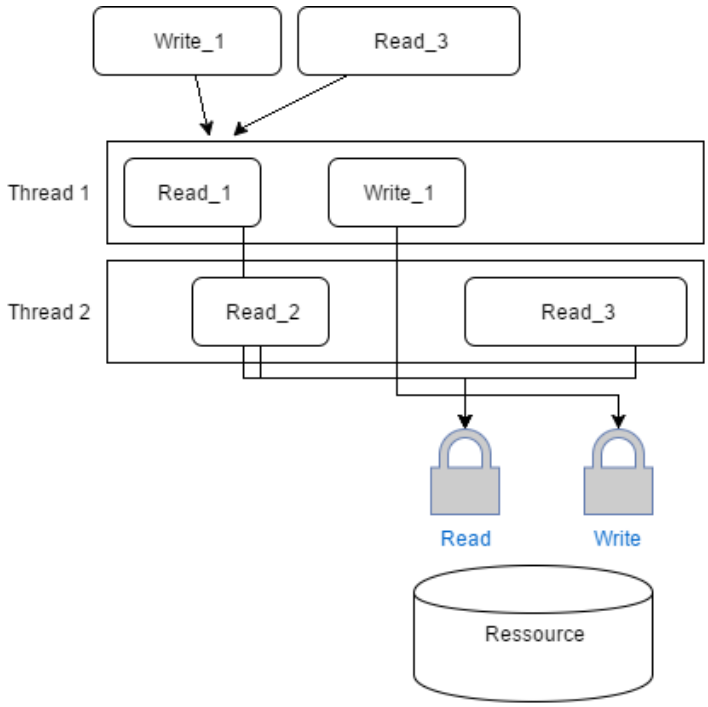

FIGURE 6.2 – Ligne du temps montrant un ordonnancement par verrou.

6.2.2 Ordonnancement par règle

L'autre possibilité qui existe pour mettre en place de l'ordonnancement est d'assigner une priorité à la règle elle-même, et non à son type d'accès. Il existe beaucoup d'algorithmes pour implémenter de l'ordonnancement, et ceux-ci vont dépendre des caractéristiques du

moteur.[14] Au vu de notre implémentation actuelle, les hypothèses suivantes ont été posées :

- Une règle en exécution ne peut être arrêtée ou mise en pause au profit d'une autre (exception faite de situations d'urgence où une règle aurait un comportement anormal, comme une boucle infinie) => ordonnancement non préemptif
- Il existe des règles qui ont un comportement périodique et d'autres qui répondent à des événements aléatoires.
- Certaines règles partagent des ressources communes => Ordonnancement de tâches dépendantes.

Afin d'illustrer les différents ordonnancements possibles, voici un état du moteur qui sera traité avec des algorithmes différents :

6.2.2.1 Etat du moteur initial

Dans cet état initial du moteur, le pool de thread contient deux threads utilisables. Ces threads sont inactifs au démarrage.

Le moteur possède 2 ressources :

1. Ressource 1. Contient un booléen.
2. Ressource 2. Contient un flux de textes. Si le flux n'est pas vide, le booléen de la ressource 1 vaut true.

La liste active contient les règles suivantes :

1. Règle 1 : Accède en lecture à la ressource 1 et réagit au booléen en initiant les règles 2 et 3.
2. Règle 2 : Accède en lecture à la ressource 2. Lit le contenu du flux.
3. Règle 3 : Accède en écriture à la ressource 3. Ajoute une ligne de texte à la fin du flux.

6.2.2.2 Ordonnancement par priorité statique

Cette manière d'ordonnancer est la plus simple : après une attribution de valeur à chaque règle, celle qui a la valeur la plus élevée est prioritaire pour être exécutée.

Cette méthode présente l'avantage de toujours donner la priorité aux règles qui nécessitent un traitement urgent. Néanmoins, le risque de ne pas exécuter les règles moins importantes est présent. Si les règles prioritaires sont trop nombreuses, il est en effet possible de bloquer l'accès aux autres. En principe ce n'est pas grave, car qui dit règle prioritaire dit règle plus importante. Toutefois, on sent intuitivement que, même si les règles moins prioritaires peuvent attendre, ce n'est peut-être pas toujours le cas, ou du moins pas indéfiniment. La section suivante montre une alternative à ce problème.

Afin d'implémenter cette fonctionnalité, il suffit de rajouter la définition d'une variable à la classe abstraite "AbstractRule", qui indique la priorité de la règle en question. L'attribution d'une valeur est alors faite par le programmeur de règles, dans la définition de la règle en question.

6.2.2.3 Ordonnancement par priorité statique avec ajustements

L'idée est ici de garder le principe de la section précédente, mais en permettant un changement du niveau de priorité d'une règle en fonction du temps. Ainsi les règles de plus faible priorité peuvent se voir gagner des niveaux de priorité en fonction du temps depuis lequel elles sont dans la liste active.

L'implémentation se fait tout aussi simplement, en rajoutant une définition de variable à la classe abstraite "AbstractRule". L'attribution d'une valeur se fait alors dans la

définition de la règle spécifique par le programmeur de règles. Cette variable indique la sensibilité de la priorité de la règle au temps écoulé.

6.2.2.4 Ordonnancement par priorité dynamique

L'ordonnancement par priorité statique donne déjà quelques bons outils à disposition du programmeur de règles. Toutefois, il est possible que celui-ci veuille gérer de manière encore plus précise l'ordonnancement de l'exécution de ses règles. En effet, il est possible que l'exécution d'une règle ait un impact sur la priorité d'autres règles. Il est donc intéressant d'implémenter un mécanisme supplémentaire permettant de changer la priorité des règles à tout moment.

En rajoutant aux règles une méthode permettant de changer la priorité actuelle, on obtient ainsi une plus grande flexibilité et une réaction plus efficace à une situation donnée. Bien entendu, cette méthode doit être synchronisée (possédant le mot-clé "synchronized") lors de ses appels.

6.2.2.5 Ordonnancement avec prévision

Quel que soit l'ordonnancement implémenté parmi les précédentes sections, il est aussi possible d'anticiper l'arrivée de règles dans la liste active. Sans rentrer dans une analyse heuristique et probabiliste de l'apparition d'un événement, le moteur possède une liste de règles en attente pour lesquelles un temps avant exécution est connu. Dès lors, il est possible par exemple de réserver un thread à une règle avec priorité élevée si celle-ci va arriver sous un temps donné.

Cette méthode étant basée sur des estimations, il est possible de jouer avec plusieurs paramètres :

1. La priorité minimale qui permet au moteur de réserver un thread pour son exécution.
2. Le temps restant avant l'insertion de la règle dans la liste active en dessous duquel la réservation est possible.
3. Ce dernier point peut être variable si on effectue une estimation du temps d'exécution des règles actuellement dans la liste active. Le moteur peut ainsi choisir d'attendre plus longtemps avant de réserver si une règle estimée rapide est dans la liste active.

6.2.3 Conclusion sur l'ordonnancement

L'ordonnancement est possible à plusieurs niveaux de complexité et sera utile ou non en fonction du programme final. Même sans ordonnancement intégré, il est possible d'implémenter son propre algorithme avec les outils de script (voir section 6.3), mais si ces fonctions d'ordonnancement existent, le programmeur de règles peut avoir un contrôle plus direct sur le déroulement de ses scénarios.

6.3 Intégrer du monitoring et de l'interaction

Dans la caractéristique édictée par M. Tennenhouse à propos de la réactivité et du temps réel [21], il est un point qui n'a pas encore été abordé. Il s'agit de l'intégration du contrôle humain dans la boucle. Placer l'être humain dans le processus de décision peut se faire à plusieurs niveaux.

Il y a évidemment le niveau le plus élevé qui va consister en l'interaction avec le système lui-même. C'est un aspect sur lequel nous ne pouvons pas agir ici, car le moteur

est générique et détaché d'une implémentation particulière. Toutefois, en y réfléchissant, il peut y avoir un niveau moins élevé accessible à l'homme. En effet, pourquoi celui-ci ne pourrait-il pas directement agir sur le comportement interne du moteur et analyser son comportement en temps réel aussi ? Il est fort probable qu'un contrôle si proche du code ne sera pas à destination de tout utilisateur, mais il peut s'agir d'un gros avantage pour le programmeur de règles et administrateur du système.

En conséquence, l'objectif va être d'effectuer une modification du moteur qui permette au programmeur de règles d'aller plus loin que l'utilisation basique du système et d'être capable de modifier son comportement sans devoir pour cela changer le code source.

Pour permettre d'agir sur le comportement du moteur sans en modifier le code, il faut nécessairement que celui-ci fasse appel à des éléments extérieurs modifiables. La possibilité la plus directe et probablement la plus simple est de définir une grande quantité de comportements configurables. Ainsi l'édition d'un fichier de configuration permet effectivement de modifier le comportement du moteur. Mais nous désirons aller plus loin que cela et être capable d'effectuer des changements en temps réel, lors de l'exécution du moteur.

En effet, un fichier de configuration nous apporterait :

— Une possibilité de modifier le comportement général du moteur au contexte d'exécution

— Une facilité de modification : il suffit en effet de modifier le fichier de configuration, ce qui se fait rapidement.

Toutefois, cette solution ne répond pas suffisamment à nos attentes :

— La modification du fichier de configuration ne peut se faire que hors connexion, il n'y a donc aucune possibilité de modification en temps réel.

— Les variables configurables seront limitées et doivent être prédéfinies et implémentées dans le moteur. Si la modification voulue n'est pas implémentée et configurable, il est impossible d'agir dessus.

Si on veut donc aller plus loin et atteindre une réelle ouverture à la modification, il va falloir donner au programmeur la possibilité de rajouter ou greffer du code au moteur existant.

Deux méthodes différentes qui permettent d'implémenter un tel mécanisme : la programmation orientée aspect (OA) et l'ajout de scripts

6.3.1 La programmation orientée aspect

La programmation OA est une méthode de programmation qui permet de séparer l'implémentation de toutes les exigences, fonctionnelles ou non, d'un logiciel. Le principe est donc de coder chaque problématique séparément et de définir leurs règles d'intégration pour les combiner en vue de former le système final.

Par rapport à l'orienté objet, cette nouvelle technique donne la possibilité aux programmeurs d'encapsuler des comportements qui affectaient de multiples classes dans des modules réutilisables. La programmation OA permet donc d'encapsuler dans un module les préoccupations qui se recoupent avec d'autres.

Pour mieux comprendre ce principe, voici la description originale faite par Gregor Kiczales et ses collègues [5]. Une préoccupation qui doit être implémentée est soit :

- Un composant si elle peut clairement être encapsulée dans un objet ou un module. Les composants sont donc, par définition, des unités fonctionnelles d'un système. Ex. : les services imprimantes, l'accès aux bases de données d'une librairie digitale.
- Un aspect si elle ne peut pas être clairement encapsulée dans un composant. Les aspects correspondent donc souvent aux exigences non-fonctionnelles d'un système. Ex. : la synchronisation, la performance et la fiabilité d'une librairie digitale.

En utilisant ces termes, on entrevoit le but de la programmation OA : aider le programmeur à séparer clairement les aspects et les composants les uns des autres en offrant des mécanismes qui permettent de les abstraire et de les composer pour obtenir le système général. On sent donc tout de suite la différence avec les langages orientés objet qui ne permettent, eux, que de séparer les composants, rendant impossible l'abstraction claire des aspects.

Avec la programmation OA, chaque problématique est implémentée de façon complètement indépendante, sans se soucier de l'existence des autres problématiques. Dans l'implémentation d'une librairie digitale en OA, les composants d'accès à la base de données ne sont absolument pas au courant qu'ils doivent être synchronisés. Le programmeur implémente l'aspect de synchronisation avec ses règles d'intégration séparément.

Source : [1]

La programmation orientée aspect possède tous les mécanismes nécessaires à la modification du code source depuis une source externe. Toutefois, l'orienté aspect à un objectif de transversalité : le but est d'extraire un aspect du code et de regrouper cette gestion à l'extérieur du moteur lui-même.

Cette approche ne sera pas suivie ici, et ce pour plusieurs raisons :

1. Il n'est pas question d'implémenter un aspect, mais de modifier des endroits précis du code. On ne parle donc pas d'ajouter au moteur des modules tels que du logging ou une gestion des exceptions, mais des modifications plus précises et localisées
2. Cela nécessite au programmeur de connaître la programmation orientée aspect, ce qui demande de l'apprentissage supplémentaire.
3. Il n'y a pas de normalisation avec l'orienté aspect, et chaque implémentation sera différente, ce qui peut mener à de la confusion, surtout que les aspects ne sont pas toujours repérables sans outils adaptés.

6.3.2 Ajout de scripts

L'autre solution est la possibilité d'intégrer des scripts au fonctionnement du moteur. L'idée est donc de permettre d'insérer des morceaux de code externes au projet lui-même à des endroits clés.

Tout d'abord, il faut repérer quels sont les endroits auxquels nous souhaitons placer un point d'entrée pour un script. Lorsque ces points d'entrée ont été identifiés, il faut qu'un appel se produise à ce moment-là à du code extérieur.

Pour faire cela, le fonctionnement du moteur a été modifié en un moteur complètement événementiel. Nous avons déjà parlé d'événementiel plus tôt pour ce qui était de la gestion des ressources extérieures, mais cette fois-ci c'est le moteur interne qui sera modifié pour être totalement événementiel. Ainsi, au lieu d'avoir une boucle principale dans le moteur qui vérifie la liste active, qui lance l'exécution de règles, etc., nous avons une boucle qui se contente d'exécuter des événements tant qu'il en reste en attente. Comme type d'événements, on va donc retrouver : choix d'une règle dans la queue, exécution d'une règle, création d'une ressource, etc.

Afin de faire en sorte que les événements soient modifiables par un script extérieur, le moteur possède désormais la possibilité d'ajouter des méthodes qui agissent sur des événements particuliers. Le système utilise une combinaison des annotations Java et des signatures des méthodes enregistrées.

Voici un squelette typique d'une classe permettant de réagir à un événement :

```
1  public class MyClass extends EventListener {
2
3    @EventHandler
4    public void functionName(Event event) {
5      //do some stuff
6    }
7  }
```

Lorsqu'une instance d'une classe implémentant "EventListener" est enregistrée, le moteur récupère toutes les méthodes de la classe possédant l'annotation "@Event-Handler". Cette méthode doit posséder un paramètre : le type d'événement auquel il faut réagir.

Dès lors, dès que l'événement en question se produit, toutes les méthodes préalablement enregistrées sont appelées avec l'événement en question.

Cette gestion des méthodes peut se faire de la manière suivante :

```
1    public class EventDispatcher {
2      private Collection<EventListener> handlers
3            = new ArrayList<EventListener>();
4
5      /**
6      * Adds an event handler.
7      */
8      public void addHandler(EventListener handler) {
9        this.handlers.add(handler);
10     }
11
12     /**
13     * Removes an event handler.
```

```
14        */
15       public void removeHandler(EventListener handler) {
16         this.handlers.remove(handler);
17       }
18
19       /**
20       * Dispatch an event to the registered handlers.
21       */
22       public Vector<AbstractRule> dispatchEvent(APEEvent event) {
23         Vector<AbstractRule> toAdd = new Vector<AbstractRule>();
24         for (EventListener handler : handlers) {
25           toAdd.addAll(dispatchEventTo(event, handler));
26         }
27         return toAdd;
28       }
29
30       /**
31       * Dispatch an event to each method of the listener that want
32       * to handle it (has the @EventHandler annotation and right
33       * parameter)
34       * @param event the event to dispatch
35       * @param handler the listener
36       * @return
37       */
38       protected Vector<AbstractRule> dispatchEventTo
39             (APEEvent event, EventListener handler) {
40         Collection<Method> methods
41               = findMatchingEventHandlerMethods(handler, event);
42         Vector<AbstractRule> toAdd = new Vector<AbstractRule>();
43         for (Method method : methods) {
44           try {
45             // Make sure the method is accessible (JDK bug ?)
46             method.setAccessible(true);
47
48             if (method.getGenericReturnType().equals(Void.TYPE))
49             method.invoke(handler, event);
50             else if (method.getGenericReturnType().getTypeName()
51                  .equals("lu.uni.fstc.proactivity
52                  .rules.AbstractRule")) {
53               AbstractRule rule = (AbstractRule) method
54                       .invoke(handler, event);
55               toAdd.add(rule);
56             }
57             else {
58               try {
59                 Vector<AbstractRule> rules
60                   = (Vector<AbstractRule>) method
61                       .invoke(handler, event);
62                 toAdd.addAll(rules);
63               } catch (Exception e) {
64                 Global_Vars.logger.warning("return type ignored
65                     from method from listener. " + e.getMessage());
66                 method.invoke(handler, event);
67               }
68             }
69           } catch (Exception e) {
70             System.err.println("Could not invoke event handler!");
71             e.printStackTrace(System.err);
```

```
72            }
73          }
74          return toAdd;
75        }
76
77        /**
78        * Find all methods from the <em>handler</em> object that must
79        * be called, based on the presence
80        * of the HandleEvent annotation.
81        */
82        private Collection<Method> findMatchingEventHandlerMethods
83            (EventListener handler, APEEvent eventName) {
84          Method[] methods = handler.getClass().getDeclaredMethods();
85          Collection<Method> result = new ArrayList<Method>();
86          for (Method method : methods) {
87            if (canHandleEvent(method, eventName)) {
88              result.add(method);
89            }
90          }
91          return result;
92        }
93
94        /**
95        * Look for the annotation values.
96        */
97        private boolean canHandleEvent(Method method,
98                            APEEvent event) {
99          EventHandler handleEventAnnotation = method
100             .getAnnotation(EventHandler.class);
101         if (handleEventAnnotation != null) {
102           if (method.getParameters().length == 1)
103           return method.getParameters()[0].getType() ==
104                           event.getClass();
105         }
106         return false;
107       }
108   }
```

Ce code est inspiré du site https ://gmarabout.wordpress.com/2010/09/23/annotation-based-event-handling-in-java/

Ce mécanisme permet maintenant d'enregistrer des classes capables de réagir à des événements du moteur, mais, pour réagir, il faut avoir les moyens de réagir. Dès lors, l'événement en question doit posséder une série de méthodes et éléments sur lesquels elle peut agir.

Lorsqu'une méthode reçoit un événement en paramètre, celle-ci peut donc :

1. Annuler l'événement (ne fonctionne que sur les événements qui se produisent avant l'action à proprement parler. Par exemple, il y a deux événements pour l'exécution d'une règle : l'événement avant son exécution, et l'événement signalant la fin de son exécution).
2. Récupérer la classe du moteur, permettant d'accéder à toute une série de méthodes propres à celui-ci
3. Récupérer le pool de thread afin d'avoir là aussi une série de méthodes à disposition.

On voit rapidement qu'agir sur les événements peut rapidement entraver le bon fonctionnement du moteur et que l'utilisation de ce principe est réservé aux utilisateurs experts. Il permet néanmoins une grande marge de manœuvre pour toute personne voulant changer des éléments. Ce système permet d'implémenter du monitoring, de changer l'ordonnancement, ainsi que d'autres fonctionnalités spécifiques.

Pour rendre tout cela facilement utilisable, on peut aussi d'ajouter la possibilité d'assembler tous ces listeners en projet, appelés plugins, que l'on peut directement associer au moteur. Ainsi, un plugin "Sandbox" pourrait par exemple posséder tous les listeners nécessaires pour annuler la création de toutes les ressources externes.

La figure 6.3 montre le plugin développé dans l'objectif de tester à la fois le système de plugin et le moteur. Il s'agit d'un outil de monitoring permettant de facilement visualiser le temps d'exécution de chaque règle ainsi que le temps d'utilisation des ressources. D'autres statistiques sont aussi récoltées, comme le temps d'attente minimum, maximum et moyen d'une règle avant son exécution, son temps d'exécution minimum, maximum et moyen, etc...

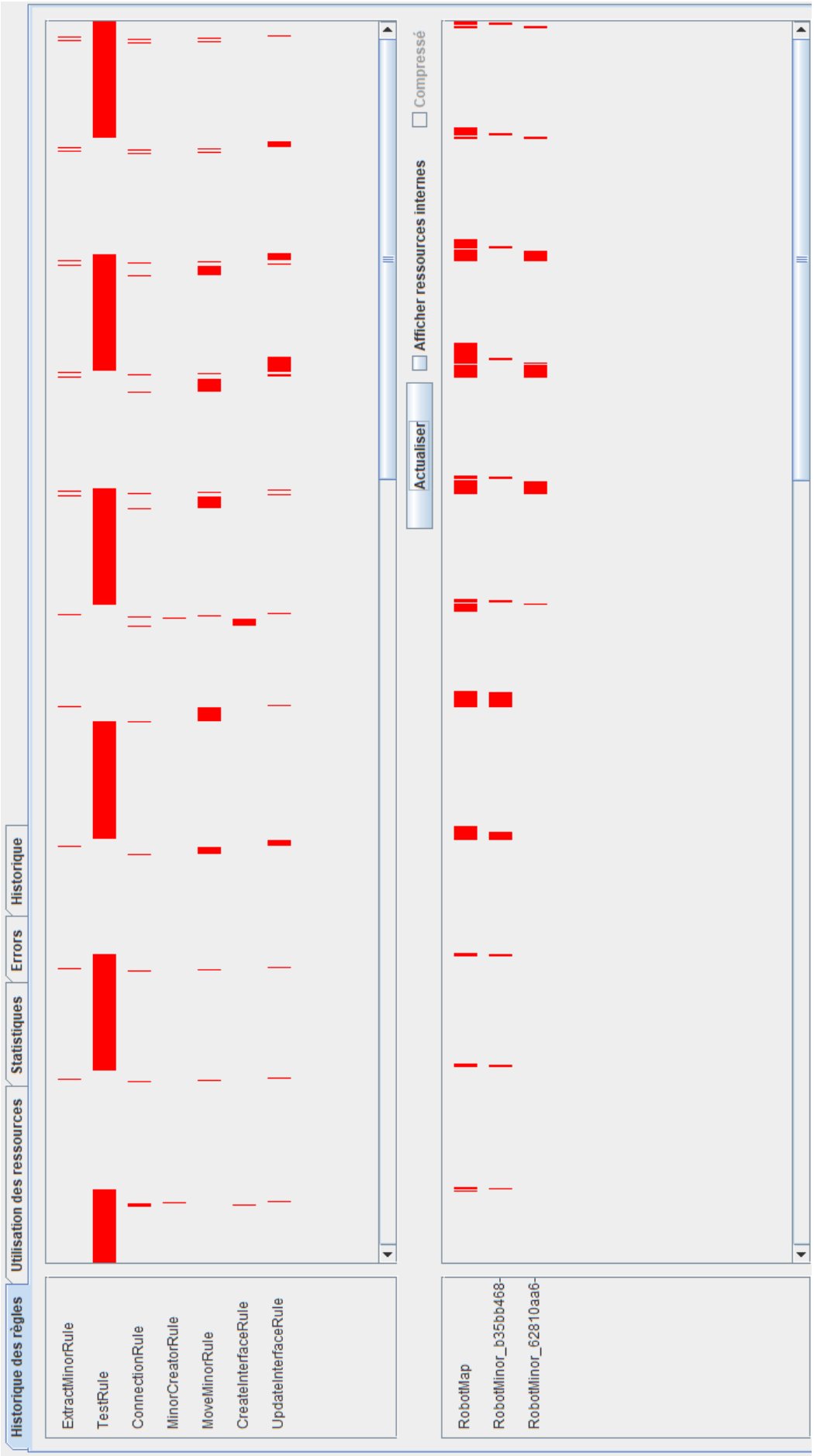

FIGURE 6.3 – Outil de monitoring utilisant le système de plugin pour tirer des informations internes au moteur proactif.

6.3.2.1 Précision sur la concurrence

Ce mécanisme de listeners, pour être capable de modifier le comportement général du moteur, est synchrone avec celui-ci. Ceci implique bien entendu que, pour conserver des performances optimales, les plugins doivent être les plus optimisés possibles.

Ainsi, dans la version d'un plugin de monitoring développé lors de mon stage, les listeners se contentent de prendre l'information, de la stocker et de rendre la main au moteur. C'est un thread externe qui récupère les informations du plugin pour les traiter. Ainsi le moteur n'est pas ralenti par le plugin.

6.4 Evaluation des améliorations

Au cours de ce chapitre, nous avons pu explorer les différents aspects et méthodes utilisables pour améliorer la réactivité du moteur proactif asynchrone. Ce chapitre nous a permis de montrer les types d'ordonnancements possibles et leurs implications. Nous avons mis en évidence qu'un système un peu plus complexe va grandement bénéficier de l'ordonnancement de ses règles, principalement en évitant des situations de famine et de verrous bloqués.

En fin de chapitre, nous sommes allés plus loin dans l'utilisation d'un moteur événementiel et avons changé le cœur même du moteur en un système similaire. Cette technique nous permet de rendre le comportement du moteur modifiable par l'extérieur et donc permet la création de plugins et de scripts capables d'influer sur le résultat final.

Ces changements ont permis de rendre le moteur adaptable aux besoins de l'utilisateur final, tout en conservant ses performances, voire même en les améliorant !

Chapitre 7

Règles et IoT - MQTT

7.1 Introduction

Comme nous l'avons vu en début de ce mémoire, pour améliorer l'efficacité de la proactivité dans un système, celle-ci a besoin du maximum d'informations sur les éléments qui l'entourent et qui forment son contexte d'exécution. Avec l'émergence actuelle de l'Internet des objets, il devient intéressant d'essayer de combiner notre moteur avec ce que l'on peut trouver actuellement dans ce domaine.

Pour le moment, l'Internet des objets apporte une liste d'avantages et d'inconvénients qu'il nous revient d'adapter à nos besoins. En effet, l'Internet des objets nous offre une grande diversité de capteurs et de données de tous types, mais, en contrepartie, on se retrouve avec des dizaines de protocoles différents, des problèmes de sécurité et de mise à jour, ou encore des problèmes de disponibilité et de performance.

7.2 Protocoles de l'IoT

Les sections suivantes passent ici en revue les protocoles utilisés en IoT. Seuls les protocoles de la couche 7 du modèle OSI seront abordés, soit la couche applicative.

7.2.1 HTTP

HTTP est un protocole très répandu, mais malheureusement celui-ci n'est pas adapté à l'Internet des objets. En effet, HTTP n'est pas réellement prévu pour :

1. Émettre des informations one-to-many.
2. Réagir à des événements lorsqu'ils se produisent.
3. Transférer de petits paquets de données en grande quantité.
4. Transmettre des informations sur des réseaux peu fiables
5. S'adapter à une situation de haut coût de transfert de données.
6. Être utilisé sur des appareils à faible capacité énergétique.
7. Répondre aux situations nécessitant de la réactivité et une réponse en temps réel.
8. Assurer la sécurité et la confidentialité lorsque HTTP est utilisé seul.
9. Évoluer en fonction des besoins du contexte.

de [8]

Dès lors, HTTP est peu utilisé dans le monde de l'IoT, nous laissons donc de côté son intégration ici. Bien évidemment, le moteur peut accepter des ressources ouvrant une connexion par HTTP, mais son implémentation reste simple au regard des ressources existantes, le sujet ne sera donc pas abordé ici.

7.2.2 UPnP

UPnP (Universal Plug-and-Play), est un protocole Peer-to-Peer. Celui-ci utilise aussi bien TCP, UDP et HTTP. Il permet d'intégrer des appareils différents sur un même réseau et peut s'utiliser sans distinction d'operating system ou de langage de programmation. Il possède plusieurs fonctionnalités, comme la découverte de services disponibles dans les appareils proches, l'obtention d'informations, le contrôle d'éléments extérieurs, la notification d'événements, et la présentation de l'appareil.

Ce protocole est principalement destiné à la cohabitation d'appareils dialoguant les uns avec les autres. Dans le moteur, il s'agit plutôt de connexions entre des appareils et un serveur (notre système). La situation d'un moteur proactif multiple dans le réseau est envisageable (voir 9).

7.2.3 CoAP

"CoAP est un protocole réseau utilisé sur des réseaux et noeuds à contraintes dans l'Internet des objets. Ce protocole est destiné à des applications machine-to-machine (M2M) comme la smart énergie ou la domotique." [4]

Sans creuser le sujet, on peut malgré tout dire que son utilisation pourrait peut-être être intéressante dans un contexte avec plusieurs moteurs proactifs embarqués dans des objets connectés. Son utilisation sous le format REST implique une intégration relativement facile à mettre en place avec l'actuel moteur proactif.

7.2.4 MQTT

Mqtt [20] est un protocole léger M2M (machine à machine) destiné à l'Internet des objets. Ce protocole est basé sur une architecture publish/subscribe. Sa simplicité lui donne la possibilité d'être embarqué sur des machines à faible capacité. Tous les clients se connectent à un serveur central (broker) qui s'occupe de rediriger tous les messages vers leurs destinataires. Ce fonctionnement est tout à fait compatible avec le moteur actuel.

7.2.5 XMPP

XMPP (Extensible Messaging and Presence Protocol), est un ensemble de protocoles destiné à la communication et à l'échange des messages (voix et visio-conférences). Il est de type client-serveur, mais, de par son type très orienté communication, il ne va pas nous servir pour collecter des données de capteurs ou autres systèmes plus typiques de l'IoT.

7.3 Le choix de MQTT

Le développement actuel du moteur est tourné vers un système où ce dernier est central et utilise une série d'objets connectés pour tirer ses informations de son environnement. Un autre pan de la recherche autour de ce moteur pourrait s'effectuer sur la discussion entre plusieurs moteurs. Dès lors, dans le présent système, il est tout à fait envisageable d'utiliser MQTT comme protocole. En effet, le moteur peut alors servir de broker et ainsi gérer l'entièreté du réseau tout en laissant une certaine autonomie aux objets en question.

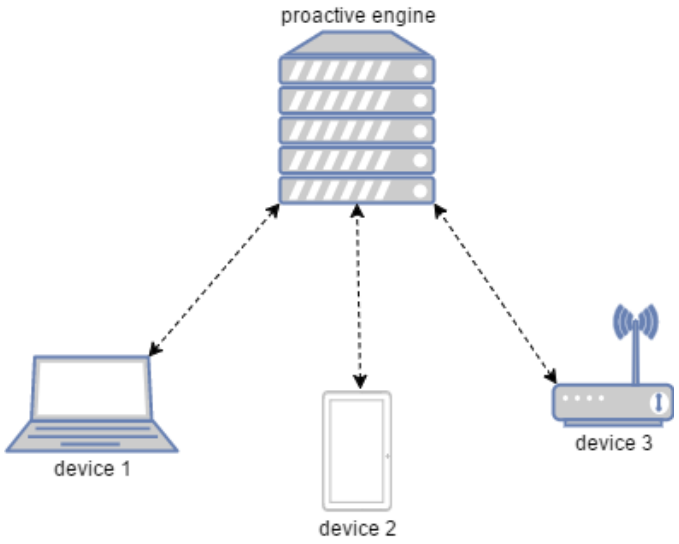

FIGURE 7.1 – Un moteur proactif et des machines connectées par MQTT

Le schéma 7.1 montre le moteur proactif connecté à 3 appareils extérieurs. Ces derniers peuvent, soit communiquer avec le moteur directement, soit envoyer un message aux autres appareils, mais toujours en passant par le moteur. Le principe de base de MQTT étant le publish/subscribe, celui-ci fonctionne de la manière suivante : un serveur contient une liste de sujets auprès desquels sont postés des messages. Différents clients peuvent écrire sur des sujets au choix. De plus, un client peut s'inscrire à un sujet et recevoir ainsi tous les messages postés sur ce sujet.

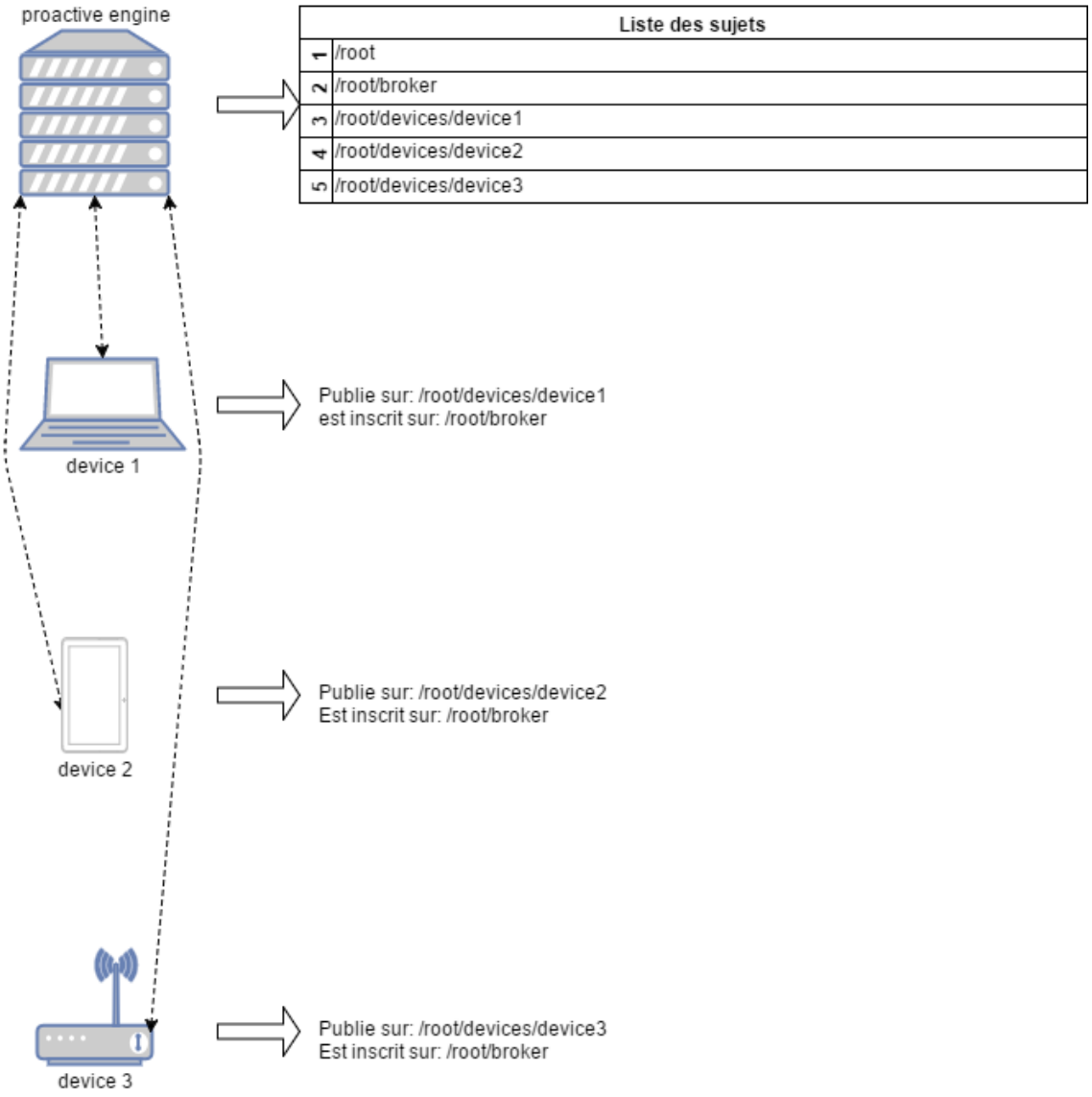

FIGURE 7.2 – Un exemple de liste de sujets et de publications

La figure 7.2 montre un exemple de table de sujets. Les sujets sont ici organisés selon une arborescence arbitraire. Le programmeur est tout à fait libre de créer l'arborescence comme il le souhaite. Dans le cas présent, "/root" sert de racine, tandis que "/root/broker" permet au serveur de publier des messages et "/root/devices/deviceN" permet aux différents devices de publier un message.

Dans une situation où les appareils discutent avec le serveur, ceux-ci écoutent le sujet "/root/broker" et le broker écoute les sujets "/root/devices/device" (qui, selon la syntaxe MQTT se note "/root/devices/*"). Toutefois, il est tout à fait possible aux appareils externes de se parler les uns les autres. Ainsi, device1 peut très bien écouter les messages postés sur "/root/devices/device2". Pour des questions de sécurité, il est possible pour le moteur d'interdire à des appareils de s'inscrire à certains sujets.

7.4 Combinaison du système de règles avec MQTT

Cette section montre une possibilité d'intégration de MQTT avec le système de règles asynchrones développé précédemment et tente de montrer que le protocole est adapté au fonctionnement du moteur proactif.

En l'état actuel, il y a d'une part un moteur basé sur un système de règle et de l'autre un protocole basé sur du publish/subscribe avec une architecture client/serveur. Dès lors, le moteur proactif va agir comme serveur MQTT. Cela permet de surveiller tous les messages qui circulent sur le réseau et d'être capable d'agir dessus. Cette intégration vise donc à implémenter un serveur MQTT proactif. Toutefois, il est bien sûr tout à fait possible de communiquer par MQTT avec un serveur distant grâce à l'application précédente.

Concrètement, le moteur va donc servir de broker MQTT. Après l'initialisation de celui-ci, le programmeur de règles va pouvoir créer des règles qui réagiront à des événements MQTT. Ces règles peuvent déjà toutes être écrites dans la version actuelle, mais il est possible d'améliorer ce fonctionnement.

En fait, pour pouvoir réagir à une activité MQTT, il faut qu'une règle puisse s'activer lors de la publication d'un message sur un topic.

Afin de permettre cette fonctionnalité, le moteur s'est vu rajouter un nouveau type de règle abstraite : l'abstractMqttRule. Cette dernière hérite d'AbstractRule et peut donc avoir un comportement classique, mais possède aussi des méthodes permettant de réagir aux événements MQTT. Ainsi, ces nouvelles règles demandent un mécanisme supplémentaire : la possibilité d'être placée dans la liste active par un élément extérieur. Dans le moteur tel qu'expliqué précédemment, une règle était, soit dans la liste active, soit dans la liste d'attente, avec un timer correspondant à sa fréquence d'actualisation. Toujours dans la version précédente, si une règle voulait activer une autre règle après son exécution, il lui suffisait de la générer. Toutefois, dans le cas présent, les règles sont en attente de la publication d'un message sur un topic et ces règles doivent donc exister avant l'arrivée du message, car aucune prise de décision n'est possible dans le cas contraire.

Voici donc dans la section suivante la solution qui a été implémentée.

7.4.1 Liste d'attente et réactions au réseau

Pour ajouter au moteur un mécanisme de réaction aux messages réseau, une nouvelle liste de règles a été ajoutée au moteur. Cette liste contient des règles qui ne seront jamais exécutées si elles ne sont pas appelées par un élément extérieur. Dès lors, le programmeur de règles peut enregistrer des instances de règles dans cette liste et les rendre réactives à des événements réseau.

Afin de permettre à une règle de réagir à un événement réseau, les Abstract-MqttRule possèdent un nom de topic auxquelles elles sont enregistrées. Pour s'enregistrer, une règle utilise les mêmes règles que MQTT dans sa syntaxe et peut donc réagir à plusieurs topics, grâce aux combinaisons de * ou de +. Lorsqu'un message arrive sur le serveur, le moteur cherche dans les règles en attente quelles sont les règles dont la cible correspond au topic. Si le topic correspond, la règle est alors placée dans la liste active.

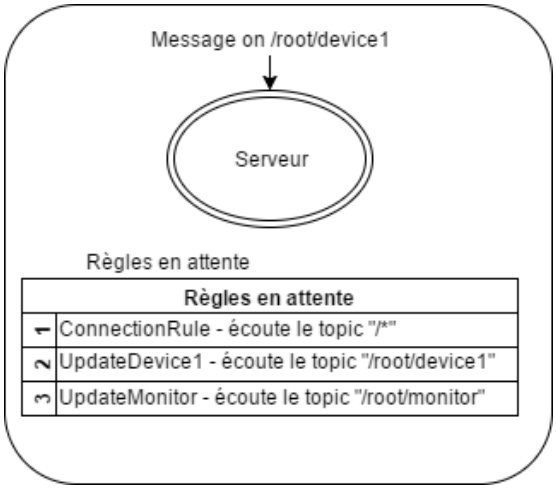

FIGURE 7.3 – L'arrivée d'un message et une liste de règles en attente

Dans la figure 7.3, un message est publié par un appareil externe sur le topic "/root/device1". Au vu de la liste des règles en attente, le moteur va pouvoir placer la règle 1 et 2 dans la liste active. Ce mécanisme permet donc au programmeur de règles d'écrire des règles capables de réagir facilement à des événements externes au moteur. Toutefois, si les messages viennent à se multiplier, il est possible que la règle qui devrait réagir à un message ne soit plus dans la liste d'attente. Deux implémentations sont possibles pour contrer ce problème. Ces deux aspects sont expliqués dans la section 7.4.2.

7.4.2 Gestion de l'ordre des messages

Les règles qui sont en attente d'un message posté sur un ou plusieurs topics peuvent être de deux types :

1. Soit la règle doit traiter les messages dans l'ordre dans lequel ils arrivent,
2. Soit la règle peut traiter les messages dans n'importe quel ordre, en parallèle.

Dans le premier cas, le moteur joue alors avec les instances de règles : lorsqu'un message arrive sur un topic concerné, la règle s'exécute normalement. Par contre, si un nouveau message arrive, et que la première règle est encore en cours d'exécution, le message est placé dans une file d'attente interne à la règle. Ainsi, lorsque sa tâche est terminée, la règle peut traiter le message suivant dans sa liste.

Dans le deuxième cas, le moteur conserve la règle dans la liste d'attente, mais crée un clone de la règle avec le message en question que le serveur exécute alors, avant de détruire l'instance en question. Ceci implique évidemment que le programmeur de règles implémente des règles clonables.

Ces deux implémentations permettent de répondre à deux besoins : certaines règles (typiquement des règles méta), sont juste là pour dispatcher les messages vers des règles plus appropriées. Ces règles peuvent s'exécuter en parallèle sans aucun problème. Par contre, des règles dialoguant avec un appareil précis voudront probablement en général conserver un ordre dans la réception des messages.

7.5 Evaluation

Au cours de ce chapitre, nous avons pu passer en revue les différents protocoles existants dans l'IoT. Nous nous sommes penchés en particulier sur le protocole MQTT pour tenter d'intégrer au mieux ce dernier au moteur. Nous avons pu montrer que ce protocole peut

être plus qu'une manière de dialoguer entre le moteur et des appareils extérieurs. Nous avons pu ainsi combiner efficacement le fonctionnement des règles et des scénarios en les rendant réactifs à des événements réseaux.

Afin de tester le système, nous avons mis en place une série d'appareils communiquant avec le moteur et une interface graphique. Pour les essais, il s'agissait de robots qui avaient pour objectif de se promener sur un terrain sans se cogner. Toutefois, ces robots sont incapables de détecter leur environnement, tout ce qu'ils ont est une communication avec le moteur proactif, qui leur sert de serveur commun. Le développement du système s'est révélé être extrêmement simple, tous les outils nécessaires étant déjà présents. Pour implémenter le système, nous avons créé deux règles dans le moteur : une première qui accepte les connexions des robots et les redirige sur un canal unique, et une deuxième règle qui communique avec chaque robot et lui indique s'il peut ou non se déplacer sur une case adjacente. Les deux règles peuvent s'exécuter en parallèle avec elles-même.

Pour pouvoir valider le déplacement de chaque robot, une ressource a été adjointe au moteur, à savoir un tableau reprenant la liste des robots et des obstacles du terrain. Durant tout le temps d'exécution du système, malgré les exécutions asynchrones des scénarios, aucun robot ne s'est jamais cogné avec un autre ni n'a eu de délai dans l'exécution de son déplacement. Nous avons pu ainsi montrer que le système peut conserver l'intégrité de ses ressources sans perdre en performances. Nous avons aussi testé le système en simulant une surcharge de travail sur des règles annexes, avec le même résultat. Les seuls délais apparents proviennent des verrous sur des ressources identiques, délais considérablement réduits avec un ordonnancement minimal ou de bonnes pratiques de programmation. Le système a montré qu'il pouvait être très efficace si le programmeur de règles garde à l'esprit qu'il travaille en asynchrone. Il est évident que pour des raisons d'intégrité des données certaines s'exécutent de manière synchrone, mais plusieurs scénarios sur des ressources différentes peuvent s'exécuter en même temps sans aucun problème.

A cet environnement nous avons aussi ajouté une interface graphique capable d'afficher en tout temps l'état du terrain sur l'écran, afin de simuler aussi des règles nécessitant un accès en lecture à la ressource, mais en continu. L'affichage n'a pas impacté le bon déroulement du programme.

Pour conclure ce chapitre, nous pouvons constater que l'intégration de MQTT au sein même du fonctionnement du moteur apporte une réelle facilité d'utilisation et une plus grande efficacité dans l'exécution de scénarios. Avec des protocoles conventionnels, il aurait été difficile de permettre par exemple à une règle de réagir aux messages des robots X et Y. En effet le protocole nous permet facilement de catégoriser les messages par genre ou destinataire, et donc d'attacher des règles à chaque catégorie ou ensemble de catégories, ce qui simplifie d'autant plus l'écriture de scénarios.

Chapitre 8

Conclusion

Nous avons commencé ce mémoire par un historique de la proactivité et de l'"autonomic computing" dans l'informatique. Nous avons montré que ces notions sont une réponse à la complexité croissante du monde informatique et en particulier de l'Internet des Objets où les appareils connectés se multiplient autour de nous. La proactivité et l'"autonomic computing" permettent de décharger l'homme des tâches de configuration, entretien, mise à jour, adaptation des différents systèmes.

Ensuite, nous avons exploré rapidement différents logiciels existants basant leur robustesse sur leur proactivité. Cet élément est mis clairement en avant et leur apporte une réelle puissance. Une machine capable de s'auto-gérer permet à l'utilisateur de faire tellement plus de choses que cela en devient un argument commercial fort. Nous avons en particulier analysé le fonctionnement du moteur proactif développé par M. Zampunieris et son équipe. Son équipe a effectivement bien compris l'intérêt de ces logiciels autonomes et elle a exploré la possibilité d'implémenter la proactivité au moyen de règles et de scénarios.

Bien que leur moteur ait permis en effet d'atteindre les objectifs voulus, à savoir réagir à des événements ou à l'absence d'événements, ainsi qu'avoir une capacité d'anticipation, la proactivité requiert aussi d'autres caractéristiques. Un moteur proactif se doit d'être attentif à son environnement, au contexte dans lequel il évolue, mais surtout être réactif, car il agit en temps réel.

Nous avons alors énoncé notre problématique. Notre objectif étant de se rapprocher d'un moteur le plus proactif possible, nous avons choisi de repartir de la base présente dans le moteur de l'équipe de M. Zampunieris. Ce moteur possède toutes les fonctionnalités nécessaires à la gestion de scénarios proactifs et au traitement de ceux-ci. Nous avons néanmoins relevé une série d'améliorations possibles. Notamment, nous avons soulevé des problèmes de réactivité et de surconsommation CPU, particulièrement dans un environnement IoT. Nous avons aussi soulevé un manque d'intégration performante avec des ressources extérieures au moteur.

Dans ce contexte, nous avons alors exploré les différentes possibilités qui s'offraient à nous pour améliorer les performances du moteur. Nous avons ainsi pointé la possibilité d'améliorer l'ordonnancement, d'exécuter le moteur dans un environnement plus événementiel et de traiter les règles en parallèle. Nous avons aussi proposé des pistes pour améliorer la communication du moteur avec des bases de données. Par la suite, nous avons montré comment mettre ces différentes pistes ensemble pour obtenir un moteur robuste et performant.

Dans cette analyse, nous avons en particulier analysé la manière dont les ressources peuvent être gérées dans un contexte asynchrone. En effet, les ressources doivent être

implémentées de manières différentes si de multiples threads y accèdent. Or, cette gestion peut se faire de plusieurs manières. Nous avons exploré alors les différentes possibilités de verrous et de ressources possibles.

Enfin, nous avons terminé l'exploration du parallélisme par l'analyse des algorithmes d'ordonnancement utilisables dans notre contexte. Nous avons vu qu'en fonction des besoins du programmeur de règles, il peut, soit utiliser peu ou pas d'ordonnancement, soit opter pour des algorithmes plus complexes. Afin de laisser les utilisateurs avancés entièrement libres dans leur implémentation, nous avons abordé la possibilité de modifier le comportement même du moteur grâce aux scripts. Ces derniers permettent de créer leur propre algorithme d'ordonnancement et leurs outils de monitoring ou toute autre fonctionnalité proche du moteur.

Dans la dernière partie, nous avons essayé de rapprocher le moteur proactif du monde de l'Internet des Objets, et nous avons passé ainsi en revue une liste de protocoles réseaux utilisés dans ce domaine. Nous avons retenu en particulier MQTT et montré comment il pouvait être intégré au système tout en tirant profit de son fonctionnement.

Au final, nous avons montré au travers de ce mémoire qu'il est possible de créer un moteur générique performant permettant à de futurs développeurs de créer leur propre système proactif.

Chapitre 9

Pistes d'amélioration

Jusqu'ici, nous avons principalement présenté comment rendre le moteur performant et réactif. Nous avons également combiné cette vision à celle de l'Internet des Objets. Toutefois, il reste des pistes d'améliorations à explorer.

9.1 Augmenter la taille des réseaux

Dans notre intégration du protocole MQTT, nous avons ouvert le moteur à la communication au monde de l'Internet des Objets. Néanmoins, cette ouverture est assez limitée : nous construisons uniquement une architecture client-serveur. MQTT peut s'avérer plus complexes, tout comme d'autres protocoles, et il pourrait tout à fait être envisageable que plusieurs moteurs proactifs se retrouvent sur ce réseau. Dès lors, il pourrait être possible d'établir une communication entre ces moteurs, par exemple au travers d'une couche réseau supplémentaire.

Intégrer un réseau doté de multiples moteurs peut apporter beaucoup d'avantages, comme la capacité de répartir les différents appareils connectés sur des points différents (répartition de charge). Plusieurs moteurs proactifs sur des machines différentes pourraient aussi se partager des tâches et déléguer l'exécution de règles à d'autres machines.

9.2 Abandonner Java

Java présente beaucoup d'avantages, comme la portabilité. Malheureusement, dans le contexte de l'Internet des Objets, exécuter un programme Java sur de petits appareils s'avère impossible, ceux-ci n'étant tout simplement pas capables d'exécuter une machine virtuelle. Or, le moteur en terme de consommation et de taille pourrait tenir (si écrit dans un autre langage) dans ce genre d'outil. Au détriment de toutes les librairies et les avantages de compatibilités que le Java peut offrir, il peut être intéressant de le coder dans un autre langage, comme le C/C++.

Annexes

Annexe A

Eléments de design

A.1 Verrou sur les méthodes héritées

Lors du développement du moteur proactif asynchrone tel qu'expliqué au cours de ce mémoire, nous avons voulu conserver une API épurée pour l'utilisateur et le programmeur de règles. En particulier, nous voulions que les méthodes réservées au moteur soient inaccessibles par le programmeur de règles. Toutefois, cela nous a posé quelques problèmes. En effet, des méthodes appartenant par exemple à AbstractRule devaient être accessibles par le moteur et une règle "fille" (abstractMQTTRule), sans être accessibles par le programmeur final.

Le projet étant développé en Java, il nous était impossible d'utiliser le mécanisme d'amitié existant par exemple en c++ qui nous aurait permis de résoudre le problème. Pour rappel, l'amitié permet à une classe d'obtenir un accès aux méthodes et attributs privés d'une autre classe. Ce système nous aurait permis de cacher les méthodes sous la mention "private" et de donner un accès aux classes nécessaires dans le moteur.

Pour garantir que seules les méthodes utilisables et sécurisées soient accessibles par le programmeur de règles, nous avons alors ajouté un système d'accès et de verrous aux méthodes réservées. Concrètement, si une classe A doit accéder aux éléments "private" d'une classe B, nous devons effectuer les tâches suivantes :

1. Créer une clé impossible à copier dans la classe A
2. Cacher les méthodes "private" de la classe B.
3. Donner un accès aux méthodes "private" aux classes possédant une clé légitime.

Afin de créer une clé unique pour une classe, nous utilisons le principe des inner class.

```
1    public static class KeyClassA {private KeyClassA(){}}
2    private static KeyClassA key = new KeyClassA();
```

Nous créons d'abord une classe publique avec un constructeur privé. Ainsi la déclaration de la classe est visible depuis l'entièreté du projet, mais impossible à instancier par une autre classe que ClassA. Nous créons ensuite une instance de cette classe, afin de servir de clé pour les appels futurs. La déclaration de cette classe est un déclaration de classe statique. Cela indique juste que la classe n'a pas accès aux données de la ClassA. N'en ayant pas besoin, cela empêche toute mauvaise utilisation.

Dans la classe B, il nous faut maintenant rendre inaccessibles les méthodes "private". En pratique, ces méthodes doivent être accessibles par le reste du projet, mais inaccessibles sans clé, et même invisibles hors du projet. Toutefois, le projet et l'extérieur de l'ensemble ne peuvent être distingués par le moteur, ce dernier ne peut donc adapter sa visibilité

entre les deux. Nous avons donc choisi de grouper toutes ces méthodes dans une inner class, elle aussi, mais qui porte un nom clair sur son utilité :

```
public class ClassB {
  public class ClassBRestrictedMethods {

    public void methodA() {
      //do some stuff
    }
  }

  private ClassBRestrictedMethods restrictedMethods =
          new ClassBRestrictedMethods();
}
```

Avec ce code, ClassB possède une instance d'une inner class contenant toutes les méthodes à protéger. ClassBRestrictedMethods n'étant pas une classe statique, elle peut accéder entièrement aux éléments de ClassB et agir dessus (il faut juste faire attention au mot-clé "this" qu'il faut remplacer par "Outer.this"). Désormais toutes ces méthodes sont sécurisées.

Pour créer un accès à ces méthodes, il faut rajouter cette méthode dans ClassB :

```
  public ClassBRestrictedMethods getRestrictedMethods
                  (ClassA.KeyClassA key) {
    if (key != null)
      return restrictedMethods;
    return null; // or send exception, log it, etc...
  }
```

Cette méthode donne accès à ClassA à toutes les méthodes privées de ClassB grâce à sa clé unique. Pour donner accès à cette méthode à d'autres classes, il suffit simplement de surcharger cette méthode avec d'autres clés en paramètres, comme on ajouterait un lien d'amitié en C++, avec ici encore plus de flexibilité sur les accès, car nous pouvons même séparer les méthodes privées en groupes différents.

Cette méthode a aussi l'avantage de supporter l'héritage ! En effet, si une ClassC hérite de ClassB, nous pouvons créer une inner class de méthodes privées héritant de celles de ClassB !

```
public ClassC {
  public class ClassCRestrictedMethods
        extends ClassBRestrictedMethods
        implements Cloneable {

    @Override
    public void methodA() {
      //this method will override methodA
      //from ClassB.ClassBRestrictedMethods
    }
  }
}
```

Cela nous a permis de protéger la totalité des méthodes sensibles du programmeur de règles. La seule méthode qu'il peut appeler qui ne lui sert à rien est "get-RestrictedMethods", car il n'a aucune clé valide. Aucune erreur de programmation suite à un mauvais appel n'est possible grâce à cette méthode et cela rend notre API d'autant plus solide.

Bibliographie

[1] Jean Baltus. *La Programmation Orientée Aspect et AspectJ Présentation et Application dans un Système Distribué*. 2001. URL : https://staff.info.unamur.be/ven/CISma/FILES/2002-baltus.pdf.

[2] Dan Bigos. *From reactive to proactive quality management with IoT*. 8 mar. 2017. URL : https://www.ibm.com/blogs/internet-of-things/quality-management-iot/.

[3] Denis Caromel. *Proactive Parallel Suite*. Sous la dir. d'ActiveEon. 19 mai 2017. URL : http://www.activeeon.com/.

[4] *CoAP*. 23 mai 2017. URL : http://coap.technology/.

[5] Kiczales G. et al. "Aspect-Oriented Programming". In : *Proceeding of the European Conference on Object-Oriented Programming (ECOOP)*. 1997.

[6] A. G. Ganek et T. A. Corbi. *The dawning of the autonomic computing era*. In : *IBM Systems Journal (Volume 42, Issue 1, 2003)*. Sous la dir. d'IBM. 2003.

[7] HP. *Services HP Proactive Care*. 29 avr. 2017. URL : http://www8.hp.com/fr/fr/business-services/it-services/proactive-care-services.html.

[8] Christian Karasiewicz. *Why HTTP is not enough for the Internet of Things*. Sous la dir. d'IBM. 9 sept. 2013. URL : https://www.ibm.com/developerworks/community/blogs/mobileblog/entry/why_http_is_not_enough_for_the_internet_of_things?lang=en.

[9] Anind K.Dey, Gregory Abowd et Daniel Salber. "A conceptual framework for supporting the rapid prototyping of context–aware applications". In : *Human-Computer Interaction* (avr. 2001).

[10] Jeffrey O. Kephart et David M. Chess ans Thomas J. *The vision of autonomic computing*. In : *Computer (Volume 36, Issue 1, Jan 2003)*. Sous la dir. d'IEEE Computer Society. 14 jan. 2003.

[11] Rainer Koschke. "Identifying and Removing Software Clones". In : *Software Evolution*. 2007.

[12] J.C.R. Licklider. "Man-computer symbiosis". In : *IRE Transactions on Human Factors in Electronics* (1960).

[13] Vincent Nouyrigat. *La nouvelle bombe des hackers*. In : *Science et Vie No1196*. Sous la dir. de Mondadori. France, mai 2017, p. 94–97.

[14] Laurent Pautet. *Ordonnancement temps réel*. Sous la dir. de Telecom Paris école nationale supérieure des communications.

[15] *PureStorage*. 29 avr. 2017. URL : https://www.purestorage.com/.

[16] Sandro Reis, Denis Shirnin et Denis Zampunieris. "Design of Proactive Scenarios and Rules for Enhanced e-Learning". In : *Proceedings of the 4th International Conference on Computer Supported Education, Porto, Portugal 16-18* (avr. 2012).

[17] Dobrican Remus-Alexandru et Zampunieris Denis. “Moving Towards a Distributed Network of Proactive, Self-Adaptive and Context-Aware Systems”. In : *Proceedings of the ADAPTIVE 2014 - The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications* (mai 2014).

[18] Charles Rich et Candace L. Sidner. “Adding a Collaborative Agent to Graphical User Interfaces”. In : *Communications of the ACM* (1996).

[19] Antti Salovaara et Antti Oulasvirta. “Six Modes of Proactive Resource Management A User-Centric Typology for Proactive Behaviors”. In : *Proceedings of the third Nordic conference on Human-computer interaction*. Tampere, Finland, oct. 2004.

[20] Andy Stanford-Clark et Arlen Nipper. *Mqtt*. 7 mai 2017. url : `http://mqtt.org/`.

[21] David Tennenhouse. “Proactive Computing”. In : *Communication of the ACM*. 2000.

[22] R. Want, T. Pering et D. Tennenhouse. *Comparing autonomic and proactive computing*. In : *IBM SYSTEM Journal, VOL 42, No 1*. Sous la dir. d’IBM. 2003.

[23] Wikipedia, éd. *Proactive Parallel Suite*. 19 mai 2017. url : `https://en.wikipedia.org/wiki/ProActive`.